SOFTWARE PROFILING VIA ELECTROMAGNETIC SIDE-CHANNEL SIGNAL

A Dissertation
Presented to
The Academic Faculty

By

Alireza Nazari

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2020

To my parents, brother and sister

for their endless love, support and encouragement from 7,000 miles away

ACKNOWLEDGEMENTS

I would like to extend my gratitude to my academic advisors, Dr. Milos Prvulovic and Dr. Alenka Zajic, for their inspiration, continuous support, and guidance throughout this thesis. This work would not be possible without their time and ideas.

I am thankful to my thesis committee, Dr. Alexander Orso, Dr. Moinuddin K. Qureshi and Dr. Tushar Krishna. Their insights, suggestions and feedbacks have greatly improved this thesis.

I would like to express my warm appreciation to all of those with whom I have had the pleasure to collaborate during this work. I would like to especially thank Nader Sehatbakhsh for his help and friendship. I also wish to thank my other collages, Moumita Dey, Frank Werner and Monjur Alam.

I would like to thank my family for everything they have done for me over the years. Their constant support, encouragement and love inspired me throughout every moment of this long journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis develops general methods to exploit information leaked in Electromagnetic (EM) emanations for profiling software applications. A broad range of computing devices and software applications can benefit from these methods. Computers radiate EM emanations when voltage and current flows change as a result of software program activity. EM emanations can be intercepted and analyzed to extract information about corresponding computation. Traditionally, EM side-channel has been leveraged to gather critical information about cryptographic algorithms. This information is used by cryptography researches to extract secret cryptographic keys from computing devices as the devices perform encryption operations. The design and implementation of this analysis is usually done ad-hoc, for a specific implementation of a cryptographic algorithm on a particular machine.

The wide range of information that can be gathered from EM emanations signals suggests that it is useful for more purposes than cryptographic analysis. Moreover, there are two major benefits in using these signals. First, they can be received remotely and no contact with device is needed. This specially benefits embedded devices where access to the device is not easy or even possible. Second, the EM signal can be received and processed in a physically separate machine. This also benefits real-time and cyber-physical devices which have very limited computation and memory resources. Until now, only few bodies of work tried to explore the complex relationship between EM emanations, underlying architecture and software application. It is viable to use EM emanation as a tool for profiling application and infer various levels of information from it. This information may span from detailed statistics of an event in the underlying machine to timing information of the software program's code in large granularity. However, profiling this information requires a general approach that can be automatically applied to diverse programs and machines. Toward this goal, this thesis has developed (1) A new approach for profiling software programs that leverages unintentional EM side-channel and allows highly accurate profiling

of loops and other repetitive activity, without perturbing the profiled system, (2) A new method for anomaly detection in program execution that monitors application's repetitive behavior, (3) an external memory profiler that infers last-level cache misses from EM side-channel signal, (4) a technique that extends the other proposed methods to multi-core systems by blind separation of EM emanation sources.

CHAPTER 1
INTRODUCTION

1.1 Motivation

Electronic circuits within computers generate electromagnetic (EM) emanations as a consequence of changes in current flow within a computing device [1, 2, 3]. It is well documented in the literature that EM emanations often contain some information about program activity in the system. Most research work on EM emanations has focused on the potential risk that they create as side-channels [2, 4], and on countermeasures against such attacks [5, 6, 7, 8, 9, 10, 11, 12, 13]. However, the wide range of information that can be gathered from such signals suggests that EM side-channel signal is potentially useful for far more purposes and its application should not be limited to hostile activities. Unfortunately, since the relation between side-channel signals and this information is conventionally perceived as *complex*, few bodies of work have tried to explore these non-hostile applications [14, 15, 16, 17]. In non-hostile scenarios, not only EM signal is not considered as a source of potential threat, but it is leveraged to provide an insight into application's behavior. Thus, while usually systems try to prevent information leakage from EM emanation by attenuating signal, e.g. shielding, or noise addition, in non-hostile scenarios no countermeasure against it presents.

There are two major advantages in using EM signal. First, it can be received remotely and no direct contact with the target device is needed. This feature can be specially critical in a scenario in which the device is not accessible, e.g. medical in-body devices. Hence, the required information can be obtained remotely.

Second, EM signal can be received and processed by a physically separate system. This means there is no need for allocation of resources, such as compute, memory or system support, in target device. Hence, no interference with its main task is required. For

some systems, especially in real-time and cyber-physical domains, program execution can change significantly when interrupts alter the performance characteristics of the program. For example, these programs often include different algorithms that are used when there is a risk of not meeting real-time deadlines or safety criteria, and instrumentation and/or interrupts may cause the system to use these "emergency" algorithms more often, leading to an execution that is no longer representative of profiling-free execution.

This detachment also potentially helps systems that have very limited resources, e.g. the processor performance and/or memory capacity may be barely sufficient for the system's primary function and cannot accommodate any overheads, the system's power source (e.g. energy harvesting) may not be able to support the added energy consumption to collect, store, process, or transmit extra data, and the processor has no advanced hardware support for secondary purposes because it would significantly add to its (extremely low) cost. Many internet-of-things (IoT) devices have both types of limitations - real-time/cyber-physical system operating under severe cost, energy, etc. limitations.

Even in some resource-rich scenarios, e.g. already-deployed systems that suffer from unexplained performance problems, tasks such as profiling are highly desirable to be done without changing the program or the system activity in any way, to ensure that actual program behavior in that deployment is captured accurately.

The key insight in drawing a relationship between application's code and EM signal is that repetitive program activity (e.g. a loop) causes the unintentional EM signals to exhibit periodicity, i.e. the spectrum of these EM signals will have "spikes" at frequencies that correspond to the time spent in each repetition of the program activity. For example, a loop whose per-iteration time is T will create a spike in the EM spectrum at frequency $f = 1/T$ and multiples (harmonics) of that frequency. Relationship is explained further in Chapter 2.

This thesis exhibits that EM emanation is a viable mean for profiling applications and explores various levels of information that can be extracted from it and their potential appli-

cation. Analysis of EM signal,whether for purpose of side-channel attack or countermeasure, is usually done application- and hardware-specific. However for non-hostile purposes, a general way to gather information is needed. Consequently, many steps that are typically done manually in side-channel attacks can be automated. The main goal of this work is to explore generalized profiling methods via EM signal where method is not bound to specific application or hardware and to show how this information is valuable.

1.2 Spectral Profiling: Observer-Effect-Free Profiling by Monitoring EM Emanations

Profiling of program execution is an essential part of performance optimization and performance analysis efforts. Typically, the goal of profiling is to identify the regions of code where the bulk of execution time is spent (”hot” regions), which allows developers or the compiler to focus their optimization efforts. Another common goal of profiling is to gain more insight into the performance characteristics of some region of code, such as typical execution time or the variation in execution time for a code fragment, which can help programmers understand a performance problem and identify potential solutions [18, 19, 20, 21].

Typically profiling is done by software instrumentation or periodic hardware sampling. However, both methods have significant overheads. First, software instrumentation requires changes in the source code or binary. Second, many of simpler processors, that are commonly used in IoT devices, are not equipped with hardware support for profiling. Third, both methods require frequent interruption of application execution whether for instrumentation or checking hardware counters. As a higher accuracy and finer profiling granularity is targeted, more overhead is forced; too a point which the execution is not representation of normal execution.

Spectral Profiling is a new approach to profiling that leverages unintentional EM emanations from the profiled systems. Spectral Profiling allows highly accurate profiling of

loops and other repetitive activity, without perturbing the profiled system, the program it runs, or the characteristics of the execution, in any way. Spectral profiling relies on training with known inputs to identify which spike frequencies are characteristic for which loop. During profiling, Spectral Profiling monitors the spectrum in real time to identify when spikes characteristic for each loop appear and disappear, allowing it to determine when each loop is entered and exited. Additionally, the frequency of the spike and its shape allow Spectral Profiling to determine the average per-iteration time for each loop, and the distribution of the per-iteration time around that average.

1.3 EDDIE: EM-Based Detection of Deviations in Program Execution

Malicious attacks typically try to exploit a vulnerability in victim system to take over control flow and execute a malicious code. In many threat scenarios, especially Advance Persistent Threats(APT), attacker aims to keep intrusion undiscovered by minimizing its effect on normal functionality or code.

Malwares detection is a vibrant area of research and many methods for hardware, software or combined detectors have been proposed. Most detection algorithms attempt to whether model malicious activity by profiling as many known malware, or model uninfected behavior by dynamically profiling normal activities such as hardware counters, system calls, control flow, etc.. Then they continuously monitor target process and test it against existing models. Any deviation from expected behavior or overlap with known malicious activity is considered as malware.

Evidently, malware detection problem resembles software profiling in many aspects. Thus, proposing a method inspired by Spectral Profiling is justifiable. Not only it eliminates overhead and observer effect which mentioned above, using EM emanation provides another prominent security feature. Conventional methods rely on the target machine itself to verify its own behavior and potentially taking the risk that the monitoring system gets taken over by the same attack. However, using EM emanation opens up an opportunity to

fully detach monitoring unit from the target machine.

EM based Detection of Deviations in Program Execution (EDDIE) proposes a detection approach that monitors the EM emanations from the monitored machine, looking for spikes in the EM spectrum that correspond to execution of loops and other repetitive activity in the program. This provides EDDIE with an information-rich aspect of the execution that it can monitor from outside of the monitored system, without any support within or cooperation from the monitored system itself. Even relatively brief injected bursts of activity (a few milliseconds) are detected by EDDIE with high accuracy, and that it also accurately detects injections of even a few instructions into an existing loop body.

1.4 EMPROF: Memory Profiling via EM-Emanation in IoT and Hand-Held Devices

Spectral Profiling is able to provide a breakdown of how execution time is spent. However, it is usually a major interest to profile microarchitectural events to get a better insight into performance contributors. Memory subsystem has been specially in spotlight because while many MLP and ILP techniques are proposed to hide memory latency, still disparity between processor and memory is a major source of performance loss. Hence, profiling Last Level Cache (LLC) misses, which is still order of magnitude faster than memory, has been a prominent interest in performance analysis. A wide range of cache profiling techniques have been documented in literature. Full-system simulations provide detailed information about the architectural events, but they are extremely slow. Also, it is immensely difficult to accurately model modern processors and their memory subsystem. Hence, in order to get more accurate information, profiling at run-time is preferred. Hardware support for such profiling is provided by performance counters and periodic interrupts to sample and attribute them to particular parts of code. This process raises the same issues that Spectral Profiling addressed.

Moreover, another shortcoming of hardware counters is that they provide aggregate numbers and do not gather information on individual misses. Some high-end proces-

sors support counters such as `stalled-cycles-frontend` and `stalledcycles-backend`, that allow measurement of the overall number of stalled cycles over a period of time, but it would be much more helpful to know the stalled time associated with each LLC miss since the purpose of LLC miss counters is to use them as a proxy of performance degradation introduce by memory subsystem. Providing detailed timing information on misses would give a better understanding of which ones are costly in terms of performance and help when trying to debug real-time behavior of many embedded and IoT systems.

EMPROF profiles memory and its impact on performance. Unlike previous memory profilers, EMPROF monitors the electromagnetic (EM) emanations produced by the processor as it executes instructions. By continuously analyzing these EM emanations, EMPROF identifies where in the signal's timeline each period of stalling begins and ends, allowing it to both identify the memory events that affect performance the most (LLC misses) and measure the actual performance impact of each such event (or overlapping group of events). Because EMPROF is completely external to the profiled system, it does not change the behavior of the profiled memory subsystem, and requires no hardware support, no memory, counter or other resources, and no instrumentation on the profiled system. It also can provide valuable accurate timing information about each individual LLC miss. Finally, EMPROF can complement Spectral Profiling [22]. So, it can be applied to the same EM signal to attribute each event/stall identified by to the loop-level regions of code (identified by Spectral Profiling) in which that event occurs.

1.5 Blind Source Separation of Electromagnetic Side-Channel Signals

All platforms that mentioned above, Spectral Profiling, EDDIE and EMProf, are targeting a single core processor. Also, the target device is running in isolation and there is no interference from any other device. Currently, severe energy and cost constraints on IoT devices has limited their computation capacity. Hence, most IoT devices are dedicated to a single functionality(e.g. smart thermostats), where a low-budget microprocessor or single

core processor is sufficient.

However, in the rise of more sophisticated IoTs which integrated multi-core processors(e.g. Apple TV A10 processor), it is extremely useful to expand Spectral Profiling capabilities to function in a multi-core environment. Not only this improves the range of IoT devices that can benefit from these platforms, but it also opens up new opportunities to use them in personal computers and servers.

We provide a proof-of-concept technique that receives mixed signals from multiple active on-chip cores and forms estimations of signals that are generated by each core. The proposed technique uses a time-frequency masking method to blindly separate EM emanations sources. This technique takes advantage of variation in leakage patterns from each on-chip core. It also relies on the observation from Spectral Profiling that the time-frequency footmarks of applications are sparse. Since the time-frequency footmarks are usually non-overlapping and each time-frequency bin is dominated by one application, a binary time-frequency mask can achieve accurate separation. Same technique also can decouple EM emanations of a target device from interferences due to other devices. This technique can be used as a preprocessing block before applying EDDIE or Spectral Profiling and enables each instance of them to monitor an application without any change in their monitoring scheme.

1.6 Research Contributions

The research contributions of this thesis are

- Spectral Profiling: A new approach for profiling that requires no profiling-related support or activity on the profiled system.
- A proof-of-concept implementation of this approach to demonstrate its feasibility in practice.
- An experimental evaluation that shows this approach achieves high profiling accu-

racy, both in a real system and in cycle-accurate simulation.

- EDDIE: a new approach to monitoring execution and detecting anomalies without any modification to or cooperation from the monitored system.
- A proof-of-concept implementation of EDDIE that demonstrates its potential.
- A detailed characterization of EDDIE implementation in the context of code injection, showing that EDDIE can detect injected code even if a brief (a few ms) burst of injected code is executed, and that it can detect injections of just a few instructions within a loop body.
- EMPROF: A new memory performance profiling method that can be applied without any impact (or even contact with) the profiled system,
- A proof-of-concept implementation of this profiling method,
- An experimental validation and evaluation of this profiler on cycle-accurate simulation and on three real-world systems.
- A proof-of-concept separation technique that enables Spectral Profiling and EDDIE to support execution of multiple applications on a multi-core machine.

1.7 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 describes a background on EM emanation side-channel and how it relates to application code. Chapter 3 describes Spectral Profiling, an observer-effect-free profiling method by monitoring EM emanation. Chapter 4 presents EDDIE, an EM-based detection of deviation in program execution. It indicates how data-rich spectra information is and how it can be applied into malware detection. Chapter 5 describes EMPROF, a memory profiling method via EM-emanation

in IoT and hand-held devices, Chapter 6, explains a blind source separation technique to extend Spectral Profiling and EDDIE to support multi-core processors.

CHAPTER 2

BACKGROUND

2.1 EM Side-Channel

It has been well documented in the literature that electronic circuits within computers generate detectable EM emanations called side-channel EM radiation [1, 2, 3]. The existence of side-channel EM radiation, and the potential risk it poses for computer security, was reported in the open literature as early as 1966 [1, 23], and much of this literature refers to the even older (classified) TEMPEST work [23, 5]. Much of the early work on EM emanations focused on information leakage created by signals from cathode-ray-tube (CRT) computer monitors [24] and on how to reduce such risks [5]. In practice, however, these risks were largely eliminated by the demise of CRT monitors, as their successors, LCD monitors, create much weaker EM fields.

Research interest in compromising EM emanations has been renewed with the mass-market introduction of smartcards (e.g., EMV "chip" credit/debit cards). A typical smartcard has a microcontroller operating at low frequencies (<300 MHz) and usually executes a single program (cryptographic authentication). EM emanations resulting from their program activity can leak information about the embedded cryptographic key(s) [2, 4], both through direct emanations that are caused by intended current flows within circuits (from switching activity while adding two numbers in a processor) and through indirect emanations caused by electromagnetic coupling among chip circuits. Numerous countermeasures have been proposed that reduce information leakage from smartcards [5, 6, 7, 8, 9, 10, 11, 12, 13, 25], including adding low-cost shielding (e.g., metal foil), using asynchronous circuits, and changing the layout of circuitry.

Attacks of this kind on high-performance (server, desktop, and laptop) systems are

more difficult because they often require capturing signals at a sampling rate much faster than the devices' clock rate, which is impractical for GHz clocks [3, 26, 27]. Despite these difficulties, it has been shown that information can be transmitted via EM emanations [28], even in the presence of significant countermeasures [3], and cryptographic keys can be extracted from modern computers using EM side-channel analysis [26].

Given these findings, it is only natural to wonder whether we can learn more about a program's behavior by observing side-channel signals. Recently, for instance, current (power) fluctuations were used to identify webpages during browsing [29] and even find anomalies in software activity [30, 14]. Results in [27, 31, 32, 33] show that differences between different instructions can be measured in EM analog signals across different devices (e.g. desktops, laptops, FPGAs) and also identify which aspects of program activity modulate which EM-emanated signals.

2.2 Spectral Footmarks of EM emanation

Among all emanated signals from a real-time system, some of the strongest and farthest propagating signals are created when an existing strong periodic signal (e.g. a clock signal) and its sidebands become stronger or weaker (Amplitude-Modulated) depending on processor or memory activity [32]. In this work we leverage these modulated unintentional EM emanations for software profiling. To the best of our knowledge, this is the first time that these unintentional EM emanations are used for software profiling purposes.

Fig. 2.1 shows a spectrum of an AM modulated loop activity. We can observe a spike in a spectrum at $f_{clock} = 1.0079$
GHz. This signal is created by the processor clock periodic activity and acts as a carrier signal in AM modulation. On the left and the right side from the carrier, we can observe two spikes. They correspond to loop activity with execution time of 77 ns ($\approx 13\ MHz$) in "bit_count" benchmark from MiBench [34] suite that is AM modulated onto a carrier clock frequency $f_{clock} = 1.0079$ GHz. Please note that two spikes in the spectrum are an

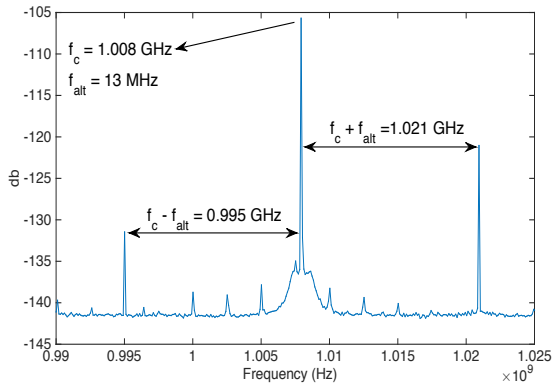


Figure 2.1: Spectrum of an AM modulated loop activity.

artifact of AM modulation. Figure 2.1 indicates that if we observe a program spectra during runtime, we can deduce which spikes correspond to current executing path(s). By knowing the frequency of each path before runtime, during execution we can deduce (1) which path in the program is currently active, (2) how much time program has spent in this path, and (3) how many times this path has been executed.

As an application goes through different functions and loops, its spectrum is expected to change over time. To capture this dynamic spectrum behavior of an application, we use a short-time Fourier transform (STFT) that is defined as [35]

$$STFT\{s(t)\}(\tau,\omega) \equiv X(\tau,\omega) = \int_{-\infty}^{+\infty} \mathrm{x(t)w(t-\tau)e}^{-j\omega t} dt, \tag{2.1}$$

and then compute a spectrogram as follows:

$$spectrogram(t,w) = |STFT(t,w)|^2 \tag{2.2}$$

In STFT, a long signal is divided into shorter, equal, and slightly overlapping segments (windows). Computing Fourier Transform of these segments separately and plotting variable spectra over time allows us to capture spectrum characteristics. For STFT, the length of windowing is of particular importance. Narrower window gives us better time resolution,

but low frequency resolution, whereas wider windows provide better frequency resolution with lower time resolution. In chapter 3 we describe methodology of proposed Spectral Profiling which exploits the spectral footmarks. In chapter 4 we describe how an extension to Spectral Profiling can be used to detect malware.

2.3 Software Profiling

There is a large body of research on collecting program profiles, as profiling can provide a substantial amount of information about the run time behavior of programs, and this information can in turn help numerous tasks. In particular, program profiling information can be useful for code optimization (e.g., [36]), testing and debugging (e.g., [37]), and software maintenance (e.g., [38]). A vast amount of work is done on developing tools and algorithms that provide information on how much time is spent in different regions of code (e.g. [18, 19, 39, 40, 41, 42]) and detect ”hot” regions of code.

State-of-the-art profilers for modern processors can be classified in two major categories: *sampling-based continuous profilers*[19, 39, 40, 41, 43] and *direct measurement* [18, 20, 44, 45] profilers. While sampling-based profilers mostly use interrupts to frequently read and sample hardware counters and extrapolating full program statistics based on sampled data, direct measurement profilers directly instrument application by adding instrumentation to code automatically or manually. Recent advancements in HW support for profiling in high-end processors enable programmers to get more information about the runtime behavior of the application with less overhead [40, 46, 47, 43, 48]. However, this overhead still could be problematic. In order to read these performance counters (e.g. Reading time stamp counter for getting the current clock cycle on an Intel processor [49]) we need to add some instrumentation in the source code or in binary file. This instrumentation is intrusive especially when the size of the region we want to profile in source code is just few instructions, and it can change the timing and behavior of the code. Sampling based profilers try to mitigate this effect by reading the performance counters less frequently and

extrapolating data at the cost of losing some accuracy.

Embedded systems and IoT devices profilers share the same concept as profilers in modern processors (sample-based, and direct-measurement). Where for direct measurement, there is more options and freedom other than just using performance counters. Few of them are Logic analyzers (e.g. [50]), Trace/Debug interfaces, and JTAG interfaces. Furthermore, recent generations of IoT devices (e.g. [51]) are able to support Linux operating systems which enables programmers and software developers to use capabilities of Linux OS (e.g Perf [43]) for profiling.

Another popular type of profilers are path profiles, which reveal more information than basic-block or edge profiles. Unfortunately, obtaining path profiles (and code profiles in general) requires code instrumentation, which is invasive and comes at the cost of high runtime overhead. The efficient path profiling algorithm proposed by Ball and Larus [52], for instance, which is an efficient (acyclic) path profiling technique that forms the basis of many other path profilers, was reported to impose an average runtime overhead of 50%, with as much as a 132% overhead in the worst case. Other studies (e.g., [53, 54]) also report similarly high overhead.

Targeted path profiling [55] is another related approach that tries to reduce the execution overhead by not instrumenting the regions in the code where information could be obtained using edge profiling. Pertinent path profiling [56] is yet another technique that addresses the high overhead problem by optimizing the data structures used for profiling. Finally, partitioned path profiling [57] proposes the idea of parallel path profiling, which profiles a program by evenly distributing the number of probes into multiple cores.

Despite all the work done so far to reduce the runtime overhead of instrumentation based program profiling, profiling still comes at a non-negligible cost in terms of overhead. Although this overhead is tolerable in some cases, it is not always so (e.g., in the case of embedded devices with limited resources or real-time systems). Moreover, instrumentation is an intrusive technique that can change some aspect of a program's dynamic behavior of

such code, especially in the case of complex, real-time, and/or multi-threaded systems.

For some systems, especially in real-time and cyber-physical domains, program execution can change significantly when instrumentation (or calling interrupts) change the performance characteristics of the program. For example, these programs often include different algorithms that are used when there is a risk of not meeting real-time deadlines or safety criteria, and profiling instrumentation and/or interrupts may cause the system to use these "emergency" algorithms more often, leading to an execution profile that is no longer representative of profiling-free execution.

Profiling is also difficult in systems that have very limited resources, e.g. the processor performance and/or memory capacity may be barely sufficient for the system's primary function and cannot accommodate profiling overheads, the system's power source (e.g. energy harvesting) may not be able to support the added energy consumption to collect, store, process, or transmit the profiling data, and the processor has no advanced hardware support for profiling because it would significantly add to its (extremely low) cost.

Unfortunately, many internet-of-things (IoT) devices have both types of limitations - real-time/cyber-physical system operating under severe cost, energy, etc. limitations. Even in some resource-rich profiling scenarios, e.g. already-deployed systems that suffer from unexplained performance problems, it would be highly desirable to profile them as-is, without changing the program or the system activity in any way, to ensure that actual program behavior in that deployment is captured during profiling.

2.4 Cache profiling

Memory profiling approaches are designed to help programmers and system developers characterize the memory behavior of a program, and this information can then be used by the compiler and/or programmer to improve performance [58, 59]. In general, simulation, hardware support, and program instrumentation are three main methods of profiling cache behavior. Cache simulation can help identify general locality problems that would cause

performance degradation on cache-based systems in general [60], but is typically much slower than native execution and fails to model the complex architectural details of modern LLCs and their interaction with memory. Hardware support typically takes the form of hardware performance counters, which are counting actual microarchitectural events as they occur without significant performance overheads due to counting itself [61]. However, an interrupt is needed to attribute a counted event to a specific part of the code, which would cause major performance degradation if every event is attributed. Instead, a sampling method is used, typically by interrupting each time the count reaches some pre-programmed threshold T, to provide statistical attribution of the events [61, 62, 63, 43]. This creates a trade-off between 1) the granularity at which events are attributed, and 2) the overhead introduced by profiling and the distortion of results by profiling activity itself [64, 65, 66]. Program instrumentation methods can also be a powerful memory profiling approach [67, 68], but they have a similar trade-off between precision/granularity and overhead/disruption.

Memory profiling based on counting events, such as LLC misses, collect information that is only a proxy for the actual performance impact of these events, and in many cases it would be more useful to account for the actual stall activity these events cause rather than the number of the events. To overcome this challenge, some high-end processors provide `stalled-cycles-frontend` and `stalled-cycles-backend` performance counters that allow measurement of the overall number of stall cycles in a period of time, but to limit performance impact and interference these are not attributed to individual LLC miss events. If such stall-time accounting could be attributed to individual LLC misses, it would provide better understanding of which misses are the most costly in terms of performance, and it would also provide tail latencies among LLC misses (misses whose latency is much higher than average), which would help when trying to debug real-time behavior of many embedded and IoT systems.

In general, a processor has two types of stalls. Front-end stalls occur when the front-end

of the processor cannot fetch instructions, e.g. because of a miss in the instruction cache (I$), and the processor is fully stalled when it completes instructions it already fetched but still cannot fetch new ones. This typically occurs for hundreds of cycles when the I$ miss is also a miss in the LLC (which is typically unified for instructions and data) and thus experiences the main memory latency. Back-end stalls occur when the processor cannot retire an instruction for a while, e.g. because of a data cache (D$) miss, and the processor is fully stalled when it runs out of one or more of its resources (such as ROB entries, load-store queue entries, etc.) and can no longer fetch new instructions. This typically occurs for hundreds of cycles when the D$ miss is also an LLC miss and thus experiences the main memory latency.

In a sophisticated out-of-order processor, the fully-stalled condition is averted for tens of cycles because the processor already has many tens of instructions in various stages of completion when an I$ miss occurs, and because it has plentiful resources when the D$ miss occurs. The exact number of cycles during which the processor can keep busy depends on the program and the processor's microarchitectural state at the point of the miss, but an LLC miss has latencies in the hundreds of cycles and thus typically still results in numerous fully-stalled cycles. Most resource-constrained devices, such as IoT and hand-held devices, use simple in-order cores that need less power and produce less heat [69, 70], but can avert a full stall for fewer cycles during an LLC miss. These simple cores are superscalar and still can benefit from ILP and MLP by dispatching and executing multiple instructions in-order and send more than one memory request on multiple read channels to multi-banked LLC. Because non-stalled cycles during a cache miss (by exploiting ILP) have far less impact on performance than stalled cycles, and because multiple cache misses that are in progress concurrently (by exploiting MLP) result in fewer overall number of stalled cycles than if these misses were handled with no overlap, accounting for the processor's stalled time rather than counting its LLC cache misses provides better insight into how (and why) LLC misses affect a program's performance. The role of this variable memory latency in

performance prediction is studied in [71, 72]. Hardware counters indicating stall cycles due to off-chip memory accesses are used to estimate this latency [73]. Our focus in this paper will be primarily on accounting for and attributing stalled cycles, but for clarity of presentation we will often use the term MISS to refer to a sequence of stalled cycles that are all caused by one LLC miss or even by several highly-overlapped LLC misses. Also, we will often use the term MISS LATENCY to refer to the number of stalled cycles that are all caused by one LLC miss or even by several highly-overlapped LLC misses.

CHAPTER 3
SPECTRAL PROFILING: OBSERVER-EFFECT-FREE PROFILING BY MONITORING EM EMANATIONS

3.1 Overview

In this chapter we present *Spectral Profiling*, a new approach to profiling that leverages unintentional EM emanations from the profiled systems. Spectral Profiling allows highly accurate profiling of loops and other repetitive activity, without perturbing the profiled system, the program it runs, or the characteristics of the execution, in any way.

The remainder of this chapter is organized as follows. Section 3.2 presents the Spectral Profiling approach and details our proof-of-concept implementation. Section 3.3 presents the evaluation setup and results for our proof-of-concept implementation of Spectral Profiling and some analysis about the runtime behavior of loops and how some architectural parameters of the profiled system affect our results.

3.2 Spectral Profiling

This section describes the Spectral Profiling approach, and our current implementation of it, in more detail. To illustrate how Spectral Profiling works, we use the "Basicmath" application from the MiBench [34] suite as a running example.

Spectral Profiling has two phases, training and profiling. In the training phase, we run the application with known training inputs to identify which spectra correspond to which part of the program (mostly loops), and also to identify the valid orderings between the parts of the program. In the profiling phase, we run the application with unknown inputs, record how the spectrum change over time, and combine that with the information from training to detect which part of the program is executing at each point in time.

Spectral Profiling's recognition of activity is based on recognizing the corresponding spectrum. Any spectrum fundamentally corresponds to the signal observed over some interval of time (window), and the duration of this window represents a trade-off between temporal resolution and frequency resolution. Temporal resolution corresponds to being able to tell where exactly some program activity begins and ends. Fundamentally, a spectrum that corresponds to some time window "blurs together" activity for the entire window, so spectra collected with a short window allow more precise identification of the time when program activity has changed. This means that, to improve temporal resolution, we should use spectra collected over very short intervals of time. However, the number of frequency bins (i.e. the frequency resolution) in the spectrum is proportional to the duration of the time interval, so a spectrum collected over a very brief interval "lumps together" similar frequencies into one frequency bin. This means that two program activities that have spectral "spikes" with different shapes and/or similar frequencies cannot be told apart when using short-window spectra because the spectrum only has one bin for the entire frequency range where both spikes are.

Thus the time window should be short enough to capture relevant events in the profiled execution, e.g. it should be shorter than the duration of most loops - intuitively, attribution of execution time to specific code in the application will be performed at the granularity that is similar to the size of the window. In our *Basicmath* benchmark example, we use a window of 1ms with 75% overlap between consecutive windows, which provides attribution with 0.25 ms granularity and precision between 0.25ms and 1ms.

Table 3.1: Measured and Calculated Frequency for loops in "Basicmath" application

Loop Number	Frequency (measured)	Frequency (calculated)
Loop 1	289.12 KHz	289.1 KHz
Loop 2	720.3 KHz	721 KHz
Loop 3	2.628 MHz	2.577 MHz
Loop 4	2.733 MHz	2.69 MHz

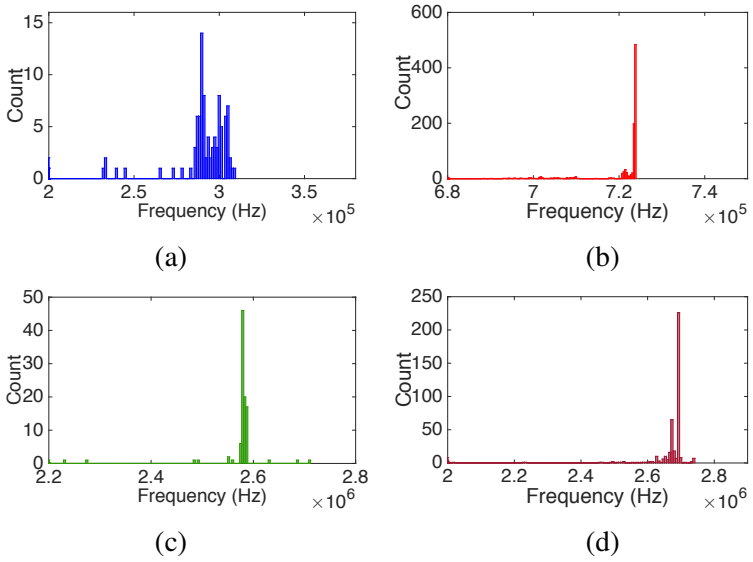


Figure 3.1: Histogram of frequencies that correspond to per-iteration execution time ($f = 1/T$) for four different loops in Basicmath benchmark for small (training run) inputs.

3.2.1 Training Phase

The goal of the training phase is to collect spectral signatures for all regions of the program, and also to identify the possible/probable sequence of the program's execution, i.e. when one region of code is executed, which regions can possibly be executed immediately after that.

When we collect spectra during training, we face a dilemma. We can run the program just like we do for profiling (no modifications to the code, no change to the system, no collection of any information on the profiled system). The spectra obtained that way will then be the same as the spectra obtained during profiling, except when spectra produced by a region of code (e.g. a loop) are input-dependent. However, when training spectra are collected this way we do not know which spectrum corresponds to which part of the code.

Alternatively, we can instrument the program (or use interrupt-based sampling) to record which part of the code executes at what time, but the spectra collected in such execution are distorted by such changes and will poorly match the corresponding spectra during profiling.

Our current approach to resolving this dilemma is to first use instrumentation to mea-

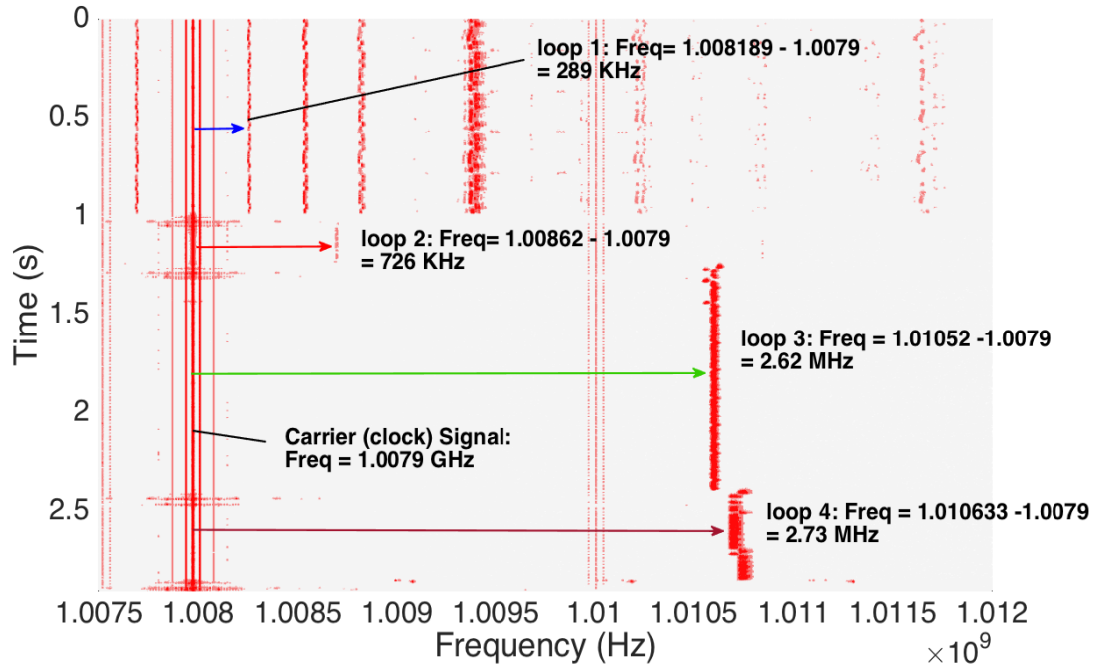


Figure 3.2: Spectrogram for *Basicmath* benchmark with large (profiling run) inputs.

sure the average per-iteration execution times for each loop, then re-run the program with the same training inputs but without the instrumentation to get the undistorted spectra, and then use the per-iteration execution times and the frequencies of spikes in the spectrum to create spectrum-to-loop mappings.

Finding Per-Iteration Execution Time

We place instrumentation at the beginning and end of the loop body, which allows us to get the current timestamps when the iteration begins and ends, subtract the two, and thus get the execution time of the loop body (for that one iteration). As the application executes, these per-iteration times are stored in memory, along with information about which loop they correspond to. When the application ends, we use this information to compute the average per-iteration execution time for each loop instance. Note that we are not interested in the total execution time of the loop, nor do we directly use this information in our profiling. The per-iteration time is only used to identify the frequency at which the corresponding

spectrum should have a spike. For example, if the average per-iteration time for a specific loop is T, we will expect the corresponding spectrum to have a spike at a frequency that is relatively close to $f = 1/T$.

In our *Basicmath* example, there are four different loops in the source code. Each of the four loops is instrumented to collect per-iteration execution time T, and Fig. 3.1 shows, for each loop, a histogram of frequencies $f = 1/T$ that correspond to these per-iteration execution times (i.e., 3.1.(a) corresponds to loop 1, 3.1.(b) corresponds to loop 2 and so on). This provides us with the approximate frequency at which to expect a spectral "spike" for each loop, along with information about the width and shape of each spike.

Finding Spectral Signatures for Each Loop

After calculating the frequency of each loop, we re-run the application with same inputs but without any instrumentation or profiling-related activity on the profiled device, and record the spectra for each time window. In each spectrum, we identify the spikes, then compare their frequencies and shapes to the histogram obtained from the previous (instrumented run). The matches are imperfect because instrumentation perturbs the execution time of a loop's iteration, and thus changes the frequency and shape in the histogram. However, our matching is highly accurate because frequencies that correspond to different loops tend to differ more than the instrumentation-induced errors, because the error introduced by instrumentation is usually in the same direction (increases the per-iteration execution time), and also because our matching approach utilizes the fact that the two runs used the same inputs and thus have the same sequence of loops. For example, Loop 3 and Loop 4 have relatively similar frequencies, but because we know that Loop 3 is likely to have a lower frequency than, and be executed before, Loop 4, the spectra corresponding to these loops can still be correctly "assigned". In addition, after successfully assigning spectra to the loops, we will also have the sequences of the "assigned" loops.

Table 3.1 shows the list of frequencies for four loops in *Basicmath*. The "measured"

column in the table shows the actual frequency of the loop (i.e. in the instrumentation-free run), and the "calculated" column shows the average frequency calculated from the instru--mentation-enabled histogram. The relative error between the calculated and measured frequency for these loops is up to 2%, but we can still easily match them. Also note that the frequency error introduced by instrumentation increases as the frequency increases. This is because instrumentation has more effect on tight loops (short time per iteration, i.e. high frequency).

After matching spectra to loops, we pre-process the spectrum that corresponds to each loop to identify the "spectral signature" for the loop. In our implementation, the signature is a list of frequencies for the strongest spikes in the spectrum, after removing spikes that appear in all spectra (e.g. for EM signals, for example, this eliminates spikes caused by radio stations, etc.). Note that the signature is not just one number that corresponds to the fundamental frequency of the loop. Some loops have a group of spikes instead of one spike, because their per-iteration execution time takes several discrete values (with some variation around each of them). In most cases, the spectrum also contains not only the spikes that correspond to the per-iteration execution time (fundamental frequency), but also spikes at multiples (harmonics) of that frequency. These additional spikes help differentiate spectra that correspond to different loops, so the signature we use includes all spikes whose magnitude is sufficiently above the noise floor.

3.2.2 Profiling Phase

In training, we identified the spectral signature for each loop, and we have also identified the possible/prob-

-able sequences of loops (essentially, which loops can execute immediately after which other loops). Profiling consists of running the application with unknown inputs and obtaining profiling information about those runs.

Matching of Loop Spectra

Because the profiling inputs are different from training inputs, it is natural to wonder if the spectrum of a loop will change. We have found that many loops, primarily innermost loops, have spectra that are nearly identical to those found in training. Intuitively, the spectrum changes when the per-iteration execution time changes, and in many loops only the number of iterations changes significantly from input to input, but the work of each iteration (and the statistics of branches and architectural events) remain similar. We call these Loops with Input-Independent Spectra (LIIS), and for these loops the spectrum can be matched to the corresponding spectrum from training.

During profiling, we use the same time window we used during training. For each time window during profiling, we obtain the spectrum for that window, identify the spikes in the spectrum (the spectral signature) and compare that signature to the signatures obtained during training. The comparison is performed by attempting to match the peaks in the profile-time and training-time signature. For each peak in the profile-time signature, we find the closest peak (according to frequency) in the training-time signature. If that closest frequency differs too much, the peak remains unmatched. If the closest frequency is very similar, the peak is counted as matched. After attempting to match each peak, the number of successfully matched peaks is used as the similarity metric between the signatures.

If the similarity is high between a profile-time spectral signature and the best-matching training-time signature of a loop, we attribute the execution during that profile-time window to that loop. For the vast majority of time windows that belong to LIIS loops, this similarity is very high and the execution is correctly attributed to the correct LIIS loop.

However, it is possible that none of the profile-time signatures matches the observed signature well enough. This happens primarily because the spectrum of some loops does change with frequency. For example, a com-
-mand-line flag may cause every iteration of the loop to take one path in one execution and a significantly different path in another execution or a set of control flows inside the loop

that can change the per-iteration execution time of the loop. For these loops, the spectrum still indicates that a loop is executing (spikes in the spectrum) and when the loop begins and ends (spikes appear at one time and disappear later) but the spectrum during profiling no longer matches any of the spectra from training.

3.2.3 Sequence-Based Matching

To attribute execution time to these Non-LIIS loops (and report their per-iteration execution time during the profiling run), we rely on the model of possible loop-level sequences constructed during profiling. Sequence-based matching begins after LIIS matching is completed for LIIS loops. The spectra from time windows that remain unmatched after LIIS matching are first clustered according to the same similarity metric we used to match LIIS loops to spectra from training, i.e. spectra that have many spikes at similar frequencies will be clustered together. At this point we have clusters where each cluster corresponds to a Non-LIIS loop, but we do not yet know which loop in the code this cluster corresponds to.

However, for each "mystery spectrum" we know that it should be matched to a region of code that is not a LIIS loop, and the LIIS loop spectra observed before and after the "mystery spectrum" tell us which loops have been executed before and after each instance of the "mystery" loop whose cluster we are considering. Fortunately, the model of the application's loop-to-loop transitions restricts the possibilities for matching so that usually only a single Non-LIIS loop remains as a possible match. When there are multiple possible matches, i.e. the "mystery spectra" in a cluster could possibly belong to more than one Non-LIIS loop, we match the cluster to the Non-LIIS loop whose training signature has the highest average similarity to the spectra in the cluster.

Fig. 3.2 shows the profiling-time spectrogram of the *Basicmath* application. The bold line at 1.008 MHz is the clock signal. The periodic program behavior amplitude-modulates this signal, and the straight lines to the right of this line represent the upper sideband of the modulated signal, i.e. they have the spectrum that coresponds to program behavior but that

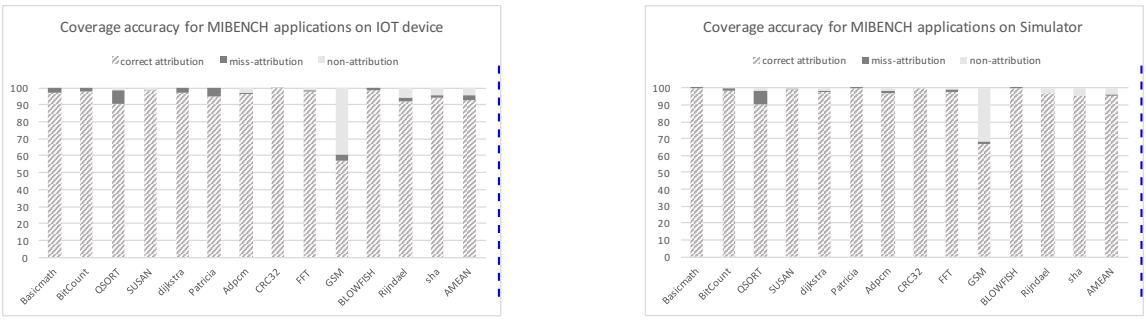


Figure 3.3: Correct attribution (striped portion) as a percentage of the overall profiled execution time.

spectrum is shifted upward in frequency by the clock frquency [32]. In this execution the four loops are executed one after the other (shown by arrows). In this case all four loops were matched through LIIS matching, but if any one of the was not matched through LIS matching, it would still be successfully matched through sequence-based matching.

3.3 Results

This section presents our experimental results, first for profiling of execution in real systems through EM emanations, and then for profiling through power signals generated through cycle-accurate simulation. We tested 13 different applications from MiBench [34] benchmark suite on both of these platforms. For real system, we used A13-OLinuXino-MICRO [51]. A13-OLinuXino is a single-board Linux computer which has ARM Cortex A8 processor [74]. For cycle-accurate simulator, we used SESC simulator [75].

3.3.1 EM-based Spectral Profiling on Real Systems

Experimental Setup

Table 3.2: Configurations for real system and simulator

Configuration	Clock Rate	Pipeline	Pipeline Width	L1 Cache	L2 Cache
Real System (Cortex A8)	1.0079 GHz	in-order	2	32 KB	256 KB
Simulator (SESC)	1.8 GHz	OoO	4	32 KB	64 MB

To demonstrate the feasibility and effectiveness of Spectral Profiling, we used it to profile applications running on a single-board computer (A13-OLinuXino), which has a 2-issue in-order ARM Cortex A8 processor with 32kB L1 and 256KB L2 caches, and uses Debian Linux as its operating system (OS). Our Spectral Profiling for this system uses electromagnetic (EM) emanations that are received by a commercial small electric antenna(PBS- E1) ([76]) that is placed next to the profiled system's processor. The antenna is placed where the clock signal has the strongest Signal-to-Noise ratio (SNR).

A spectrum analyzer (Agilent MXA N9020A) is then used to record the spectra of the signals collected by the antenna. A spectrum analyzer can be relatively costly (several tens of thousands of dollars), but we elected to use a spectrum analyzer primarily because it provides calibrated measurements, and already has support for automating measurements and for saving and analyzing measured results. In additional experiments, we observed similar spectra with less expensive (¡$5,000) commercial software-defined radio receivers.

Measurement-Based Results

We apply Spectral Profiling to all 13 applications from the automotive, communications, network, and security categories in the MiBench suite. We used a 1 ms window size with 75% of overlap between windows in all applications except GSM, where we used a 0.5 ms window to improve temporal resolution for attribution of execution time for short-lived loops.

For training, we manually insert markers before and after each loop of these applications, and each marker reads and records the current clock cycle count from the *ARM Performance Counter Unit* (ARM-PMU)[48], which provides information similar to the x86 "rdtsc" instruction [49]. The training runs are repeated with several different command line flags, in order to identify the sequence of loops that can occur in each application. We note that insertion of markers and identification of possible sequences can both be accomplished automatically by a compiler (identification of loop nests and their connectedness in

the control flow graph), but our automated compiler-based marker insertion and loop-level sequence graph was not yet ready at the time of submission of this work.

After training, for which we used the small input set [34], we perform actual profiling with the original unmodified code (no markers) and with the large input set [34]. The accuracy we measure is defined as the fraction of execution time for which our method correctly identifies the loop that is currently executing. This accuracy is not 100% because of (1) ***miss-attribution***, during which our algorithm matches the spectrum to a different loop (i.e. loop A is actually executing, but the algorithm matches the spectrum to loop B instead) at loop B, and (2) ***non-attribution***, during which our algorithm finds that the spectrum is too different from loop spectra observed in training, so it leaves such intervals un-attributed. Non-attribution is typically a result of computation whose spectrum varies widely depending on inputs, or activity that has no recognizable spectral signature (e.g. loops whose per-iteration time varies a lot from iteration to iteration).

Fig. 3.3 (left) shows the breakdown of profiled execution time into time that was accurately attributed, time that was miss-attributed, and non-attributed time. In all benchmarks except GSM, our method provides correct attribution during at least 90% of the execution time, with the arithmetic mean at 93%. Miss-attribution occurs during less than 4% of the execution time, except in QSORT, where miss-attribution occurs during 8% of the time. The larger miss-attribution for QSORT occurs primarily because the std::qsort library function does not have a stable signature, so it is often miss-attributed. It is quite possible that the variation in std::qsort spectra is a result of having multiple loops in that function. Unfortunately, our manual marker insertion did not include library code so our scheme treats the entire std::qsort function as a single entity and expects it to have the same spectrum throughout its execution. We expect that this problem can be overcome with compiler-supported loop instrumentation that includes library code. Overall, the arithmetic mean for miss-recognition is 2.26%.

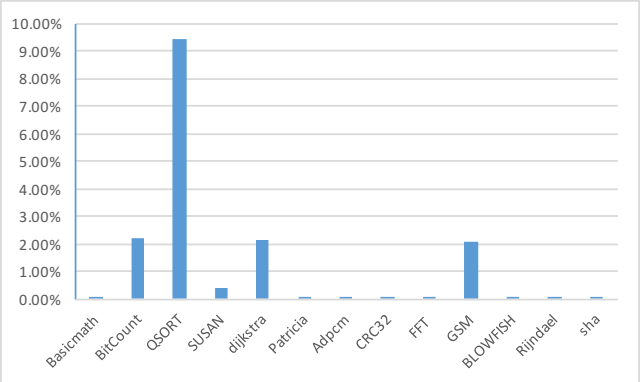


Figure 3.4: Standard error for loop start/end times, normalized to loop duration.

3.3.2 Simulated Results

To confirm that the ability to do Spectral Profiling is a result of a fundamental connection between repetitive program behavior and periodic physical signals produced as a side-effect of that computation, we apply Spectral Profiling to the spectra of power signals produced through cycle-accurate architectural simulation in SESC [75], a cycle accurate simulator which integrated CACTI [77] and WATTCH [78] power models. Table 3.2 provides more details for the simulated configuration. We used this configuration to show that even in completely different configuration, i.e. different clock rate (1.008 GHz vs. 1.8 GHz), different pipeline (In-order vs Out-of-order), etc., Spectral Profiling is still effective and is not machine or architecture dependent.

Fig. 3.3 (right) shows the accuracy results for the same applications and with the same breakdown we used for the real-system results. The accurate attribution percentage has an arithmetic mean of 98% for our simulated results, which is slightly higher than for our real-system results. The main reasons for this higher accuracy are that simulation-produced power signals are free of noise, and that they have a single-cycle resolution. In contrast to that, the EM signals received from real systems are subjected to radio-frequency noise, measurement error, and frequency-dependent distortion (for some EM frequencies the real system acts as a better "transmitter" than for others). The arithmetic mean for miss-recognition in this case is 1.19% which is slightly better than for the real system.

3.3.3 Loops with Input-independent Spectra

As discussed in previous section, some loops produce spectral "spikes" whose frequency does not change significantly with changing inputs, while others have input-dependent spectra. Recall that the frequencies of the "spikes" depend on the loop's average per-iteration time and not on the number or iteration executed in the loop, so loops with input-independent spectra (which we abbreviate as LIIS) tend to be innermost loops. Conversely, loops with input-dependent spectra tend to be outer loops whose inner loops have input-dependent iteration counts, or loops that have a set of control flows which can change the per-iteration execution time of the loop. The reason we are interested in LIIS loops is that, during profiling, their execution can be directly recognized from the spectrum. The remaining loops may still be correctly attributed, but that attribution consists of (1) identifying stable spectral patterns (the spectrum has "spikes" that remain the same for a while, which indicates that the system is likely executing a single loop) and (2) using the program's possible loop-level sequences (learned from training), i.e. the knowledge of which loops may possibly execute immediately after other loops, to attribute that activity to a specific non-LIIS loop.

Fig. 3.5 shows how much of the profiled execution time is attributed through each of these mechanisms (LIIS and Sequence). On average, 82% of the execution time is attributed through LIIS, and some applications spend nearly all of their execution time in LIIS loops. However, in several applications (especially Susan and SHA) most of the execution time is attributed through the Sequence mechanism, and almost all of this attribution is correct (see Fig. 3.3). In general, we observed that Sequence-based attribution of profiled time is highly accurate, as long as the execution contains enough LIIS recognitions to constrain the set candidate loops to which the (non-LIIS) spectrum may possibly be attributed.

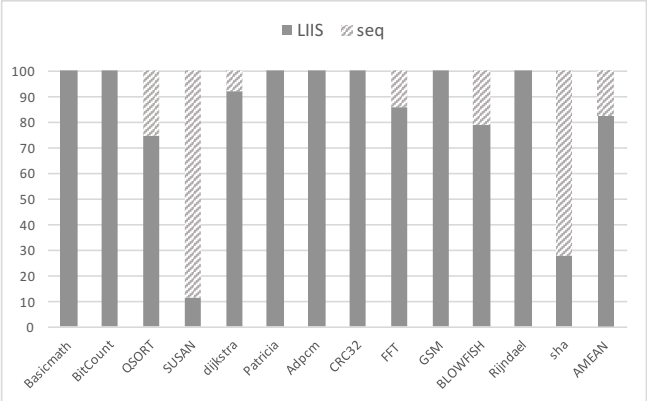


Figure 3.5: Profiled time attributed through LIIS and Sequence mechanisms.

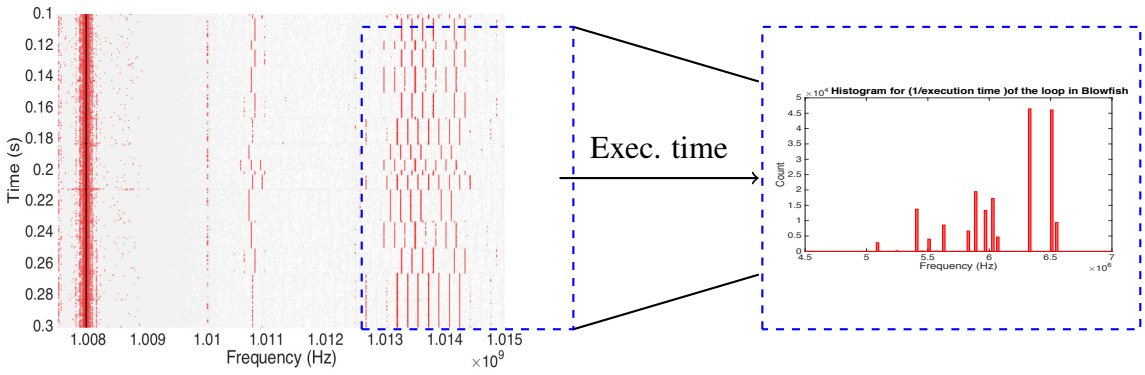


Figure 3.6: Spectrogram and per-iteration execution time for a loop in *Blowfish*.

3.3.4 Accuracy for Loop Exit/Entry Time Profiling

In addition to overall accuracy with which execution time is attributed to specific loops, we measured how accurate our method is at determining the exact time when a loop is entered and exited. Specifically, if in the profiled run some loop "Loop1" starts at time t_1, and our Spectral Profiling implementation identifies t_1' as the start time, then the difference between t_1' and t_1 is the deviation (error) for this "sample" in this experiment. The samples in this experiment are loop start and end times for all dynamic instances of all loops executed in the application, and for each application we report the standard deviation across these samples, normalized to the average duration of the loop. Because we need the actual start/end times for each loop to compute the error, we only perform this measurement in simulation (where

Table 3.3: Configurations used in simulation for Section 4.6

Configuration	Clock Rate	Pipeline	Pipeline Width	L1 Cache	L2 Cache
A(Simple Processor)	50 MHz	In-Order	1	4 KB	-
B(Modern Processor)	1.8 GHz	OoO	4	32 KB	64 MB

we can get the actual loop entry/exit times without changing the timing of the execution itself). The per-application results of this experiment are shown in Fig. 3.4, and across all benchmarks, the average of these normalized standard deviations is 1.42%.

3.3.5 Runtime Behavior of Loops

In addition to providing useful information about which regions of the code are hot and how much time is spent in each region, Spectral Profiling can also exploit the shape of the spikes in the spectrum to tell us the runtime behavior of the each loop. For example, sharp spikes indicate that almost all iterations of the loop take same amount of time. Conversely, having a wide spike, or a group of spikes, indicates that different iterations of the loop have different execution times. Such variation in per-iteration execution time could occur in outer loops when the number of iterations in their inner loops varies, and even in inner loops due to architectural events (e.g. cache misses, branch miss-predictions, etc) or differences in control flow among loop iterations. Identifying loops with unusually large per-information performance variation may help programmers identify performance problems, e.g. problems caused by unexpectedly large number of architectural events, unexpectedly frequent use of a long and seemingly unlikely program path within the loop body, etc.

To illustrate how Spectral Profiling can help understand performance of a loop, Fig. 3.6 shows the spectrogram (how the spectrum changes over time, where the spectrum is displayed horizontally and elapsed time is shown from bottom to top) for one loop in the ”Blowfish” benchmark, for the real system EM signal without any markers. We also show a histogram of actual per-iteration execution times in this loop, obtained by the markers during the execution with same inputs with markers. As seen in this figure, the duration

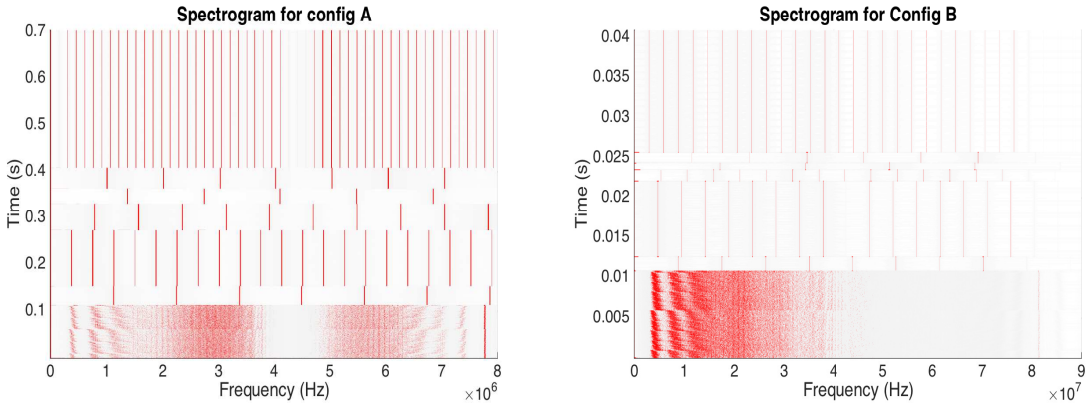


Figure 3.7: Spectrograms from two different system configurations for *Bitcnt* benchmark for large size input.

and intensity of spikes in the frequency spectrum indicate how often (and when during the loop) different per-iteration execution times occur. In this loop, the variation in per-iteration execution time is caused by cache misses on one memory access instruction and branch mispredictions on two difficult-to-predict branch instruction in the loop body.

3.3.6 Effects of Changing Architecture

To show that the ability to benefit from Spectral Profiling is architecture independent, Fig. 3.7 shows the spectrogram for a same application, using the same inputs, on different simulated systems (shown in Table 3.3). In this run, the application executes seven loops. The first loop is a nested loop that's why its signature is poorly defined at lower frequencies (which correspond to to the outer loop) with a sharp spike that corresponds to the inner loop at around 7.8 MHz in Config A and 81MHz in Config B. The remaining six loops each have a well-defined frequency, and can be clearly seen in spectrograms for both Config A and Config B. The vertical lines in the Config B spectrogram appear weaker because the spectral power of each spectral spike is distributed across a narrow frequency band as the out-of-order execution engine introduces slight variation among per-iteration execution times in each loop. However, spikes are still easily identifiable in both spectrograms, and allow us to attribute execution time to each of these seven loops, and also to determine their

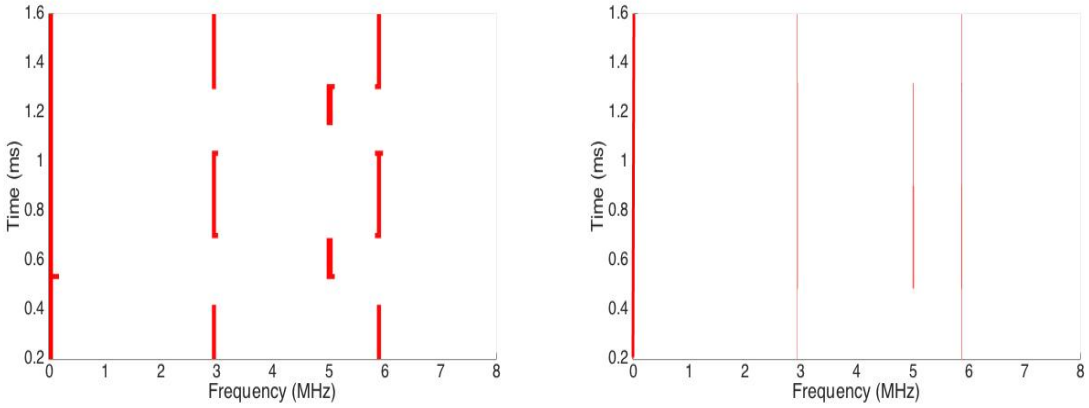


Figure 3.8: Spectrograms from two different size of window for *Blowfish* benchmark for large size input. Left figure is for 50us window, and right is for 500us window.

per-iteration execution time and its variation.

3.3.7 Size of Window

Ideally, the profiled periodic activity lasts much longer than the window we use to compute the spectrum, so the "blurring" at the beginning and end of the activity introduces an error that is negligible relative to the duration of the activity. However, as described in Section 3.2, windows that are too short provide low spectral resolution, i.e. they make it more difficult to tell different spectra apart. Consequently, the window size is a compromise between these two considerations. To illustrate this, Fig. 3.8 shows the spectrogram when the window is 500 μs and when it is 50 μs. The application spends most of its time in two loops, one with a frequency close to 3 MHz (the spectrogram also shows its harmonic that is close to 6 MHz), and the other loop has a frequency close to 5 MHz. Both spectrograms are derived from simulation-based power signals for the same simulation run. As can be seen in the figure, the shorter window allows us to clearly identify transitions between these loops. The longer window, however, sometimes (e.g. for 0.6ms to 1.2ms on the spectrogram) only indicates that both loops are active during the interval. However, note that the vertical lines in the short-window spectrogram are thicker – this indicates that the frequency bins are wider, i.e. there is less spectral resolution. In this particular example, the

frequencies of the loops are far apart, so even the 50 μs window provides enough spectral resolution to distinguish them. This indicates that the accuracy of Spectral Profiling can likely be improved by dynamically choosing the window size[1].

3.4 Summary

This chapter presented Spectral Profiling, a new method for profiling program execution without instrumenting or otherwise affecting the profiled system. Spectral Profiling monitors EM emanations unintentionally produced by the profiled system, looking for spectral "spikes" produced by periodic program activity (e.g. loops). This allows Spectral Profiling to determine which parts of the program have executed at what time and, by analyzing the frequency and shape of the spectral "spike", obtain additional information such as the per-iteration execution time of a loop. The key advantage of Spectral Profiling is that it can monitor a system as-is, without program instrumentation, system activity, etc. associated with the profiling itself, i.e. it completely eliminates the "Observer's Effect" and allows profiling of programs whose execution is performance-dependent and/or programs that run on even the simplest embedded systems that have no resources or support for profiling. We evaluate the effectiveness of Spectral Profiling by applying it to several benchmarks from MiBench suite on a real system, and also on a cycle-accurate simulator. Our experimental results show that our current implementation of Spectral Profiling on average correctly attributes 93% of execution time when applied to EM emanations from an actual IoT device, and we confirm the versatility of the approach by also successfully applying it to the power signal produced through cycle-accurate simulation of several different architectures, from sophisticated out-of-order cores to simple in-order cores. Additionally, our finding confirm that Spectral Profiling yields additional useful information about the runtime behavior of loops. Overall, Spectral Profiling can be used for profiling in systems where profiling infrastructure is not available, or where profiling overheads may perturb the results too much

[1] Recall that our accuracy results are based on experiments where the window size is constant.

(“Observer’s Effect”).

CHAPTER 4
EDDIE: EM-BASED DETECTION OF DEVIATIONS IN PROGRAM EXECUTION

4.1 Overview

Typically, an attack to IoT devices exploits a vulnerability in the software, e.g. a memory-related one [79], to take over (hijack) control flow and execute its malicious code. The most common malware detectors dynamically monitor the execution of the application, looking for suspicious activity. Some dynamic detectors look for dynamic behaviors that correspond to known types of attacks, but detection of previously unknown attacks typically relies on creating a model of its correct behavior and then looking for deviations from this model. Ideally, a detector would have a model that exactly specifies all possible correct behaviors and only correct behaviors, and then any deviation from this model can be reported as a problem.To model the normal behavior, most dynamic malware detectors use various software/hardware activities such as system calls, function calls, control flow, data flow, performance counters, etc. [80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95].

Clearly the process of modeling the normal behavior considerably overlaps with software profiling and faces the same challenges. Monitoring such information by the target machine creates both performance and resource (e.g. memory) overheads on the monitored system. Moreover, relying on the monitored system itself to actively participate in its own verification opens up new venerabilities and provides opportunities for attackers.

To overcome these problems, in this chapter we propose *EM based Detection of Deviations in Program Execution (EDDIE)*, a detection approach that monitors the EM emanations from the monitored machine, looking for spikes in the EM spectrum that correspond

to execution of loops and other repetitive activity in the program.

The overall idea of EDDIE is to use the observed EM spectra over time as a surrogate for program behavior over time, gather training data about what the EM spectra should look like in each part of the program during correct execution, and then monitor execution by looking for situations where the observed EM spectra statistically deviate from expected spectra, i.e., the observed spectra are unlikely to be outcome of a correct execution.

EDDIE obtains the EM signal from an antenna, uses the Short-Term Fourier Transform to convert this continuous signal into a sequence of overlapping windows, and then converts the signal in each window into its spectrum, which we call Short-Term Spectrum (STS). All of the actual training and monitoring in EDDIE is done on this sequence of Short-Term Spectra (STSs). Training in EDDIE consists of obtaining a number of STSs for each loop nest and each loop-to-loop transition that is possible during a valid execution in the program. During monitoring, EDDIE compares the observed STSs to those obtained during training, and reports a problem when the observed sequence of STSs is unlikely to have been produced by a valid execution.

Our evaluation focuses on how sensitive EDDIE is to the amount of injected execution, i.e. how many instructions can the injected code execute before being detected by EDDIE. We evaluate both a burst of injected execution, when the attack's injected work is performed all at once, and an injection into an existing loop, e.g. to improve stealth by spreading the injected work over time by interleaving it with the original work of the application's loop. We find that even relatively brief injected bursts of activity (a few milliseconds) are detected by EDDIE with high accuracy, and that it also accurately detects injections of even a few instructions into an existing loop body. One way to interpret these results is that, to avoid detection by EDDIE, the amount of injected execution per second must be very low, i.e. the injected code can avoid detection only if its activity utilizes a very small percentage of the monitored system's performance potential.

The rest of this chapter is organized as follows. Section 4.2 provides details of our

proof-of-concept implementation of EDDIE, Section 4.3 presents the results of our experimental evaluation of EDDIE and Section 4.4 concludes our paper.

4.2 Implementation of EDDIE

This section describes our proof-of-concept implementation of EDDIE. The overall implementation of the EDDIE prototype consists of a training phase and a detection (monitoring) phase.

4.2.1 Training

The training phase begins with a static analysis of an application to identify the loops and their possible orderings. This is done in LLVM [96] by adding a pass that builds the loop-level control flow graph of the application. Next, we run the application on the device with known (training) inputs. For each such run, we collect the resulting EM signal using an antenna or an EM probe, and we convert that signal into a sequence of sample spectra, where each spectral sample corresponds to a 1 ms window of time and overlaps 75% with the previous sample. These runs are lightly instrumented to record the time at which each loop is entered and exited, which allows us to categorize (label) each spectral sample obtained from training either as belonging entirely to a particular loop, or as containing at least some non-loop execution from a particular part of the code. For example, a run in which 4 different loops are executed will have 9 categories of spectral samples: one for execution before the first loop is entered, four for loops (one for each loop), three for loop-to-loop transitions, and one for execution after the last loop is exited.

After categorizing spectral samples, we preprocess the samples in each category to extract their features. The features we use are all the spikes (energy concentrations) in the spectrum that contain at least 1% of the total energy of the spectrum. Note that different categories will have different numbers of such features, e.g. the spectral samples for loop A might have three spikes that are strong enough to be selected as features, whereas spectral

samples for loop B might have only one spike that is strong enough. It is also possible for a category to have no features (no spikes that were strong enough).

In our experiments, we have found that spectral samples that correspond to a loop, tend to have several features (prominent spikes). Moreover, we have found that spectral samples in a loop-to-loop transition also tend to have a number of features – specifically, they tend to have features from both the loop that precedes and the loop that follows the transition. This can be explained by the fact that non-repetitive code executes briefly before a repetitive region (loop) is encountered, e.g. even a relatively inexpensive modern processor typically executes hundreds of millions of instructions per second, so repetition-free execution that covers an entire 1 ms window requires several megabytes of executable code (to avoid looping or otherwise repeating any of the instructions). For EDDIE spectra, this means that spectral samples that include non-loop activity tend to also include at least some loop activity, e.g. spectral samples that are categorized as a loop-to-loop transition are likely to include some activity (and thus spectral signature) from the exited loop, the spectral signature for the actual transition, and then some activity (and thus spectral signature) from the entered loop. This is beneficial to EDDIE because, even if the non-loop activity has a poorly defined spectral signature (because strong features in the spectrum tend to correspond to repetitive activity), the spectral samples collected for such activity are likely to contain strong and distinct signatures. For example, the spectral samples for a transition from loop A to loop B will have spectral features of both loops, so it can be distinguished from loop-A-only samples (which have no loop-B features) and loop-B-only samples (which have no loop-A features).

4.2.2 Monitoring Phase

The detection phase consists of running the application on the device with unknown inputs and without any instrumentation, collecting the EM signal, converting the signal into spectral samples, extracting the features from these samples, and finally using a statistical

test to decide (at a chosen confidence level) if the new sample is unlikely to come from the same statistical distribution as the training-time samples that are valid at that point in the execution. If there is enough confidence that the sample is unlikely to come from a valid execution, the algorithm flags that sample as anomalous.

The nature of the actual distribution of spectral samples for a given category (e.g. a specific loop in the code) can be different among different categories, e.g. one loop might have an almost perfectly Gaussian distribution, another loop have a more Poisson-like distribution, and a loop-to-loop transition might have a distribution that is difficult to match to any of the well-known distribution types. To accommodate this, we use the two-sample variant of the *Kolmogorov-Smirnov* (K-S) test [97], a well-known non-parametric statistical test. The non-parametric nature of the test allows us to test how unlikely it is that two groups of spectral samples were drawn randomly from the same population, i.e. that they belong to the same distribution, without making any a-priori assumptions about the nature of that distribution (e.g. Gaussian, Poisson, etc.).

A key parameter in our statistical test is the number of consecutive detection-time spectral samples that are used in the test (window size). Let us use n_d to denote that number. When n_d is very small, the test may fail to achieve enough confidence for reporting even when the samples come from execution of malware (i.e. we are more likely to have a false negative). However, when n_d is increased, detection tends to occur with a higher latency with respect to when the malware activity begins. When the activity has just begun, the most recent n_d samples still mostly come from valid activity (with only a few malware-afflicted samples), so the K-S test is less likely to detect the difference between that group and the training-time group (that only includes valid activity). As more malware-afflicted samples are recorded, the set of most recent n_d samples increasingly differs from training-time samples, and the larger size of n_d provides more confidence that this difference is statistically significant. This means that, at the cost of increased detection latency, high-confidence correct detection is more likely when n_d is increased. Note that for a given

application, n_d can be experimentally found during training by incrementally increasing it until false positive reaches to zero.

The detection algorithm starts with taking first n_d samples of the monitored signal and testing it against the training-time samples for the first region and continues to do so (by adding a new sample and removing the oldest one) until the test does not pass which means either the malicious activity has begun or the code has switched to another region (i.e. loop-to-loop or next loop). The algorithm then tracks the valid flow of the program to find the next possible valid region. For that, we use compile-time information about the possible valid sequences of loops, and at detection time EDDIE tracks which node or edge in that graph corresponds to the most recently observed signal. When there is more than one immediate-future possibility for a given starting point, we apply the K-S test to each valid (possible given the sequence) set of training-time spectral samples. For example, when the "current" point is loop A, the valid possibilities might be to begin A-to-B loop-to-loop transition, or to begin A-to-C loop-to-loop transition, or to begin loop B or begin loop C. If all of these valid possibilities are rejected by the K-S test, i.e., if the actual detection-time execution is unlikely to correspond to any of the possible valid behaviors at this point in the execution, EDDIE reports execution of malware. Otherwise, nothing is reported, the new "current" point is selected to be the most likely of the valid possibilities (those that were not rejected by the K-S test), and the detection algorithm continues – the next spectral sample is obtained and the updated detection-time set of n_d samples is tested against the training-time samples that correspond to valid possibilities for the updated "current" point in the execution.

4.3 Results

In this section we first present our experimental setup (Section 4.3.1), then we present results of applying EDDIE to the EM emanations from a real IoT system (Section 4.3.2). We continue with a presentation of results of applying EDDIE to the power signal produced

through architectural simulation (Section 4.3.3) and an investigation of which architectural features affect EDDIE's detection performance. Next, Section 4.3.4 provides an in-depth analysis of how EDDIE's detection performance changes as we change the percentage of loop iterations that are contaminated by code injection.

4.3.1 Experimental Setup

We use two different experimental setups. One is using a real IoT prototype system, and is a single-board Linux computer (A13-OLinuXino-MICRO [74]) with a 2-issue in-order ARM Cortex A8 processor with a 32kB L1 and a 256kB L2 cache, with a Debian Linux operating system. The EM signals emanated from this system are received by a commercial small electric antenna (PBS-E1[76]) that is placed right above the device's processor, and the signal is recorded using a Keysight DSOS804A oscilloscope [98]. While this oscilloscope is relatively expensive (several tens of thousands US dollars), note that we use it mainly because of its built-in features for automated and calibrated measurements and ability for displaying the real-time signals. In additional experiments, we have observed similar EM spectra with less expensive ($< \$5,000$) commercial software-defined radio receivers and we believe that EDDIE can work efficiently on such lower-cost setups.

Our second setup is based on the SESC [75] cycle-accurate simulator, and uses the simulator-generated power signal for EDDIE's analysis. This setup is used to confirm that EDDIE is applicable across a wide range of systems, and to gain insight into which architectural features affect EDDIE's detection performance.

In our experiments, we use a total of 10 benchmarks from the MiBench [99] suite to test EDDIE algorithm. For the real IoT system, we execute each benchmark 25 times during training. The code for the training runs contains with our light-weight instrumentation, which is implemented as a Clang tool, and the code is also subjected to a separate analysis (which is not used to actually generate code) in LLVM [96] where we added a pass that statically finds the regions and the possible transitions between regions. For monitoring in

this setup we use 25 runs per benchmark, without any instrumentation and with different inputs.

In simulation-based experiments we use fewer runs (10 training and 10 monitoring runs per benchmark) to reduce the overall simulation time.

Table 4.1: Accuracy for EDDIE monitoring of an actual IoT device

Benchmark	Detection Latency (ms)	False positives (%)	Accuracy (%)	Coverage (%)
Bitcount	42	0.99	100	99.9
Basicmath	25	1.8	99.9	99.9
Susan	32	1.39	92.1	95.9
Dijkstra	25	1.08	99.9	99.7
Patricia	28	0.98	92.3	95.2
GSM	24	0.9	96.2	57.1
FFT	17	0.76	93	99
Sha	11	1.9	97.2	98.9
Rijndael	12	0.56	99.9	97.1
Stringsearch	11	0.19	99.9	99.9

4.3.2 EDDIE Results for Measured EM Emanations of a Real IoT Device

In this set of experiments, we inject code into different regions of each application. The injections are different for loop and inter-loop regions. Injections outside loops consists of invoking a shell and then, without doing anything else, returning back to the original application. This injection results in executing 476k injected instructions and adding about 3 ms to the execution time. When injection is made in a loop, we add an 8-instruction code that consists of 4 integer operations and 4 memory accesses. The rationale for the shellcode injection is that shellcode execution is often a fundamental step in many attacks, and our empty-shellcode injection results in less injected-code execution than any real shellcode-based attack where the attack's intended activity (payload) must either be executed or at least set up within the shellcode-invoked shell. The rationale for injecting only 8 instructions into a loop body is that an injection into a loop allows the injected code to be executed repeatedly, allowing the attacker to perform significant work over time but improve stealth

by performing the work in small chunks.

The results for the IoT system are shown in Table 4.1. The first column shows the application, and the remaining columns report EDDIE's detection latency, false positive and accuracy percentages, and coverage. The results were obtained using $reportThreshold = 3$ in EDDIE's algorithm , i.e. EDDIE tolerates up to 3 consecutive K-S test rejections and only reports an anomaly for a rejection that is part of a 4-long (or longer) streak of test rejections. The average detection latency is measured as the average, among all injections that are reported, of the difference between when execution of injected code begins and the time when EDDIE reports it. This latency mainly reflects the number of STSs that are used in the K-S test. False positives are the number of STS groups that are reported as anomalous but do not contain any injected execution, as a percentage of all STS groups. The average for false positives is $<1\%$ and the highest false positive percentage was only 1.9% (for the *Sha* benchmark). Accuracy is computed for each region as the total number of STS groups with a correct reporting outcome, i.e. those that contain injections and are reported by EDDIE plus those that contain no injections and are not reported, expressed as a percentage of all STS groups. The accuracy shown for each benchmark is the average of its per-region accuracy results. On average, EDDIE's accuracy is 95%. We observed that the bulk of the inaccuracies come from borders between two regions (i.e outside the loops), and further investigation has revealed two main causes for this: (i) non-loop code during some transitions creates poorly defined peaks, so better consideration of diffuse spectral features may improve EDDIE's accuracy, and (ii) the actual inter-loop transition is usually very brief and for different executions occurs at a different point in the window on which the STS was computed, so better identification of the boundaries of the actual inter-loop transition may help to create STSs that better represent the transition. Finally, we define coverage as the amount of time during which the STS is attributed to the region in the code that actually produced it. The main reason for imperfect coverage in our implementation is that some loops have no peaks in their STSs. For example, about 40% of the execution time

in *GSM* is spent in one such loop, and this accounts for nearly all of its poor coverage.

4.3.3 Simulation Results and Sensitivity to Processor Architecture

To gain more confidence that EDDIE is a broadly applicable approach, and to get more insight into which aspects of the system's architecture have an effect on EDDIE's accuracy, we apply EDDIE to the power consumption signal generated by the SESC simulator with integrated CACTI [100] and WATTCH [101] power models for its cache and configurable pipeline. We first model a 1.8 GHz 4-issue out-of-order core with 32KB L1 and 64MB L2 caches, the power signal provided to EDDIE is sampled every 20 cycles, and EDDIE's STFT uses 0.1ms windows with 50% overlap. The code injection in these simulations is implemented by directly injecting dynamic instructions into the simulated instruction stream without changing the application's code or using any architectural registers. This maximizes the injection's stealth and is an idealized representative of an attack that uses only registers that are dead at the injection point in the original application.

Table 4.2: EDDIE's latency and accuracy when using a simulator-generated power signal

Benchmark	Average Latency	False Rejection	Accuracy	Coverage
Bitcount	7ms	0.8%	99.9%	99.9%
Basicmath	8ms	0.2%	99.9%	100%
Susan	5ms	0.7%	91.4%	96.6%
Dijkstra	10ms	0.3%	97.02%	99.9%
Patricia	13ms	0.4%	94.14%	98%
GSM	6ms	0%	100%	68.3%
FFT	5ms	0.4%	97.8%	99.1%
Sha	0.4ms	1.83%	100%	100%
Rijndael	0.6ms	0.24%	97.1%	97.2%
Stringsearch	0.2ms	0%	100%	100%

EDDIE's results for these simulation-based experiments are shown in Table 4.2. False rejections occur on average in 0.7% STSs, an expected improvement over real-system experiments because the simulation has no signal noise, no interrupts or other system activity,

Figure 4.1: Detection latency of 15 different regions in in-order and out-of-order architecture.

etc. By comparing results from simulation and real-system experiments, we can also conclude that EDDIE's accuracy and detection latency are more affected by the applications itself (i.e. mostly the shape of the spectrum for the code regions) than by factors such as noise, interference, etc.

Intuitively, we expect EDDIE to perform better on systems whose architecture introduces less variation in executions of the same region of code. To get more insight into which architectural parameters have a significant impact on EDDIE's detection performance, we configure the simulator to model an in-order processor with 3 different issue widths (1,2, and 4) and 2 different pipeline depths, and an out-of-order processor with 3 issue widths (1,2,4), 3 pipeline depths, and 5 ROB sizes, for a total of 51 configurations. We then simulate execution of 3 benchmarks (*Basicmath*, *Bitcounts*, and *Susan*) on each configuration and use N-way analysis of variance (ANOVA) to determine which factors have a significant impact on EDDIE's results.

We found that for out-of-order and in-order architectures EDDIE achieves similar false rejection and accuracy results, but its latency (i.e. number of STSs that need to be considered in the K-S test) is significantly higher for out-of-order architectures (see Figure 4.1) because an out-of-order core tends to produce more variation in its dynamically constructed instruction schedule, creating more variation among STSs and thus requiring more STSs to capture their distribution.

We also found that in in-order architectures pipeline depth and issue width have no

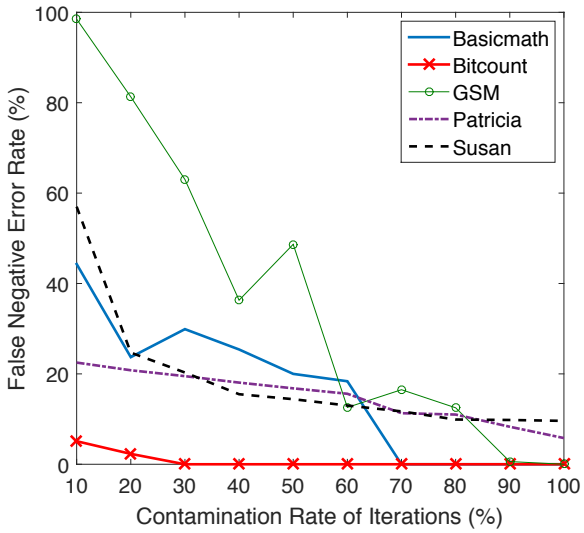


Figure 4.2: False negative rate of variable injection rates.

statistically significant effect on EDDIE's results, and that in out-of-order architectures the ROB size and issue width also have no statistically significant impact on EDDIE's results. However, in out-of-order processors pipeline depth has a weak but statistically significant impact on detection latency. A closer look at the data reveals that in 27% of the code regions pipeline length increases detection delay, and that these affected regions are all loops with control flow variation among iterations, so the likely explanation for increased detection delay in EDDIE is that a deeper pipeline results in more timing variation due to branch mispredictions, that in turn increases the size of the STS group that representatively captures this variation (and the n for the K-S test).

Finally, we repeated this analysis for different amounts of injected execution, and found that the impact of pipeline depth in out-of-order processors on EDDIE's results diminishes as the injection size increases, and for large-enough injections the pipeline depth no longer has a statistically significant impact of EDDIE's detection latency. This means that large amounts of injected activity can be detected quickly even when the processor's pipeline is deep, but for smaller injections longer pipelines result in longer detection latency.

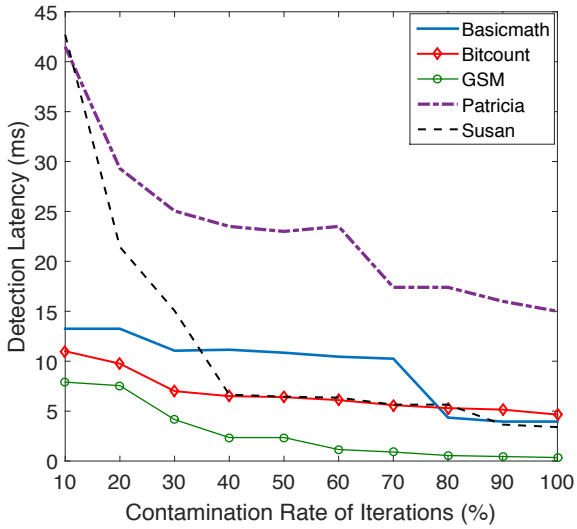
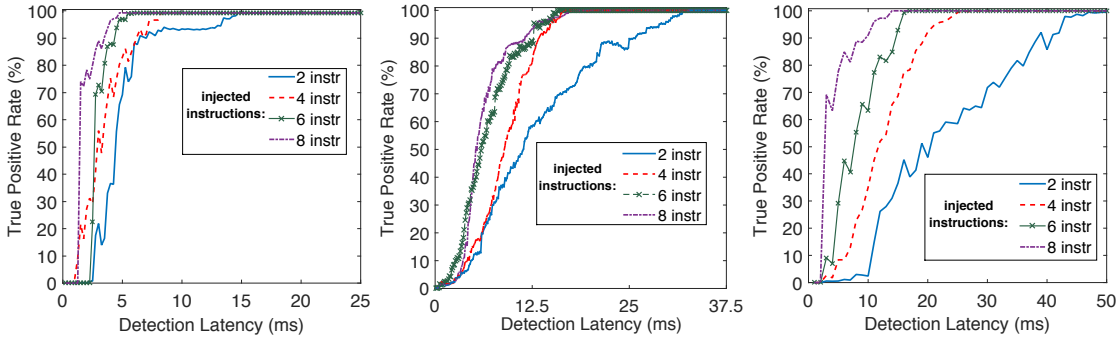


Figure 4.3: Detection latency of variable injection rates.



(a) (b) (c)

Figure 4.4: EDDIE's accuracy when changing the number of injected instructions inside loops.

4.3.4 Effect of the Execution Rate of Injected Code

An intuitive way to improve stealth is to further diffuse injected execution by injecting the code inside a loop body such that only some loop iterations execute (a small amount of) the injected code. To evaluate EDDIE in this context, we use our simulator-based setup and for the targeted loop region randomly choose the iterations that will be injected with 8 memory instructions and 8 integer operations. We use *contamination rate* to refer to the percentage of iterations that contain injected execution, and we repeat this set of experiments for contamination rates between 100% (where every iteration is injected) and

10% (where 90% of the iterations are injection-free).

Figure 4.2 shows the *false negatives*, i.e. the percentage of injection-containing STSs that are not reported by EDDIE, for different contamination rates. As expected, EDDIE's ability to detect the injection does diminish with the injection's contamination rate, but for most applications EDDIE still retains significant ability to detect injections even at low contamination rates. For example, for *Bitcount* EDDIE still detects >90% of injection-containing STSs even when only 10% of loop iterations actually contain injected execution. However, in *GSM* EDDIE detects only 5% of the STS at the 10% contamination rate. Note that this does not mean that EDDIE is inherently unable to detect injections that have low contamination rates. Indeed, Figure 4.3 shows the results in terms of detection latency (which is increased by increasing n in EDDIE's K-S test) that is needed to maintain EDDIE's accuracy. This indicates that EDDIE can very accurately detect even low-contamination-rate injections, but that detection of low-contamination-rate injections will have a longer latency.

4.4 Summary

This chapter described EM-Based Detection of Deviations in Program Execution (EDDIE), a new method for detecting anomalies in program execution, such as malware and other code injection, without introducing any overheads, adding any hardware support, changing any software, or using any resources on the monitored system itself. Monitoring with EDDIE involves receiving electromagnetic (EM) emanations that are emitted as a side effect of execution on the monitored system, and it relies on peaks in the EM spectrum that are produced as a result of periodic (e.g. loop) activity in the monitored execution. During training, EDDIE characterizes normal execution behavior in terms of peaks in the EM spectrum that are observed at various points in the program execution, but it does not need any characterization of the virus or other code that might later be injected. During monitoring, EDDIE identifies peaks in the observed EM spectrum, and compares these peaks to those

learned during training. Since EDDIE requires no resources on the monitored machine and no changes to the monitored software, it is especially well suited for security monitoring of embedded and IoT devices. We evaluate EDDIE on a real IoT system and in a cycle-accurate simulator, and find that even relatively brief injected bursts of activity (a few milliseconds) are detected by EDDIE with high accuracy, and that it also accurately detects when even a few instructions are injected into an existing loop within the application.

CHAPTER 5
EMPROF: MEMORY PROFILING VIA EM-EMANATION IN IOT AND HAND-HELD DEVICES

5.1 Overview

In this chapter we introduce EMPROF, a new approach to profile memory and its impact on performance. Unlike previous profiling approaches, EMPROF monitors the electromagnetic (EM) emanations produced by the processor as it executes instructions. By continuously analyzing these EM emanations, EMPROF identifies where in the signal's timeline each period of stalling begins and ends, allowing it to both identify the memory events that affect performance the most (LLC misses) and measure the actual performance impact of each such event (or overlapping group of events). Because EMPROF is completely external to the profiled system, it does not change the behavior of the profiled system in any way, and requires no hardware support, no memory or other resources, and no instrumentation on the profiled system. Finally, we show how EMPROF and Spectral Profiling [22] can be applied to the same EM signal to attribute each event/stall identified by EMPROF to the loop-level regions of code (identified by Spectral Profiling) in which that event occurs.

The remainder of this chapter is organized as follows: Section 5.2 provides an overview of how memory accesses are reflected in the side-channel signal, Section 5.3 describes our proof-of-concept implementation of EMPROF and Section 5.6 presents some concluding remarks.

5.2 Memory Footmark in Side-Channel Signal

This section reviews the types of stalls that we have observed in the side-channel signals. We begin with signals obtained by modeling power consumption in a cycle-accurate archi-

tectural simulator SESC [75], because the simulator can provide us with the ground-truth information about the exact cycle in which the miss is detected, in which the resulting stall begins, and in which the stall ends. We model a 4-wide in-order processor, with two levels of caches with random replacement policies, which mimics the behavior of the processors encountered in many IoT and hand-held devices. We collect the average power consumption for each 20-cycle interval, which corresponds to a 50 MHz sampling rate for a 1 GHz processor.

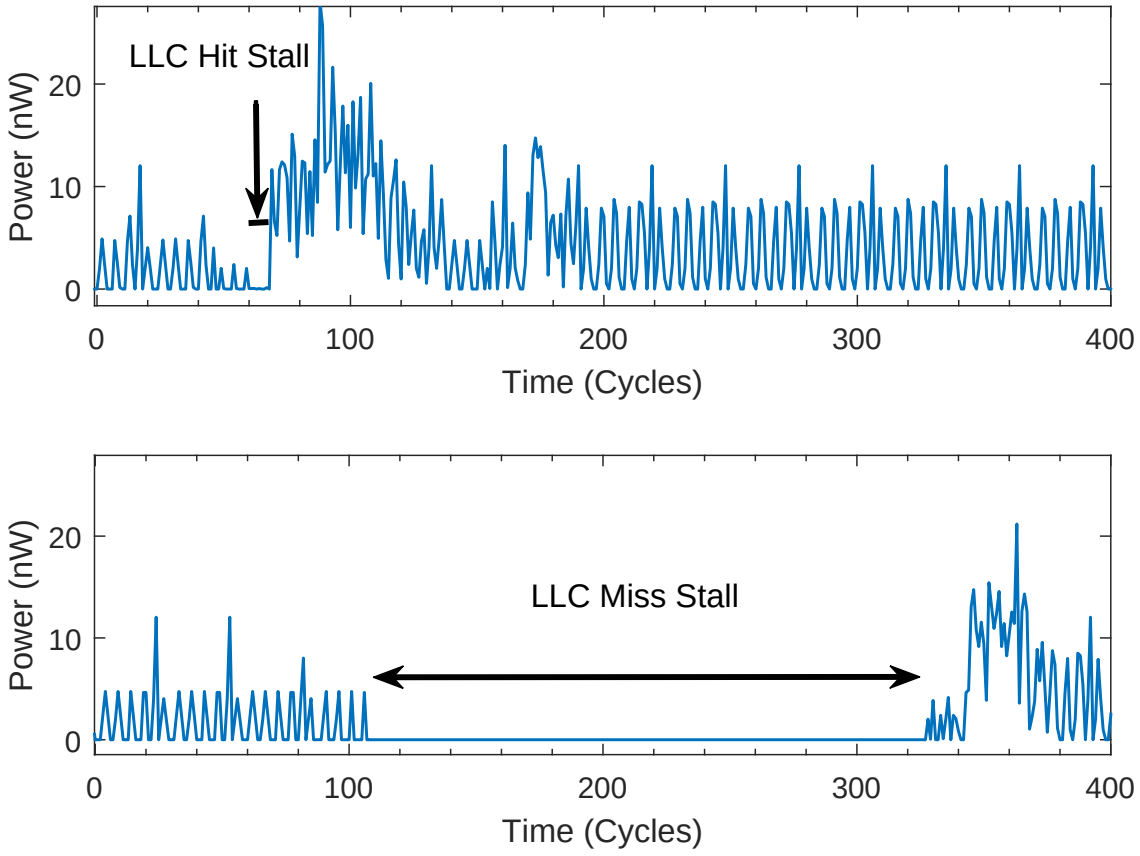


Figure 5.1: Stalls due to LLC hit and LLC miss in SESC simulator. The stall due to LLC miss is much longer than the one due to a hit.

5.2.1 Memory Access in Simulated Side-channel Signal

To observe how the side-channel signal is affected by different types of cache misses, a small application was created that performs loads from different cache lines in an array. The size of the array can be changed in order to produce cache misses in different levels of the cache hierarchy. Two signals that correspond to the same part of the code are shown in Figure 5.1. The signal that corresponds to an L1 D$ miss that is an LLC hit contains a very brief stall, during which the processor core consumes very little power because none of its major units are active. In contrast, the signal that corresponds to an LLC miss contains an order-of-magnitude longer low-power-consumption period that corresponds to a much longer stall.

When the sampling rate is reduced, each data point (sample) in the signal corresponds to an increased number of cycles, and thus shorter stalls become increasingly difficult to find in the signal. The long stalls that correspond to LLC misses, however, still contain multiple signal samples that allow the stall to be identified. The reduction in the signal's sampling rate does, however, reduce the resolution with which the duration of those stalls can be measured in the signal - e.g. with one signal sample per 20 processor cycles, the duration of the stall can be "read" from the signal only in 20-cycle increments.

We have confirmed that the power signal produced by SESC drops to its full-stall level only a number of cycles after the miss occurs, after the processor runs out of *useful* work. As explained before, this occurs either when the processor suffers an I$ miss and eventually completes the instructions it did fetch prior to that, or when the processor has a data cache miss, finishes all the work that could be done without freeing any pipeline resources, and eventually cannot fetch any more instructions until the miss "drains" out of the pipeline.

Figure 5.2 shows a signal that corresponds to several overlapping LLC misses. When the first miss occurs, the processor still has a lot of resources and its activity continues largely unaffected by the miss. During this activity, several other LLC misses occur and eventually the first miss ends while the processor is still busy and has not stalled yet. This

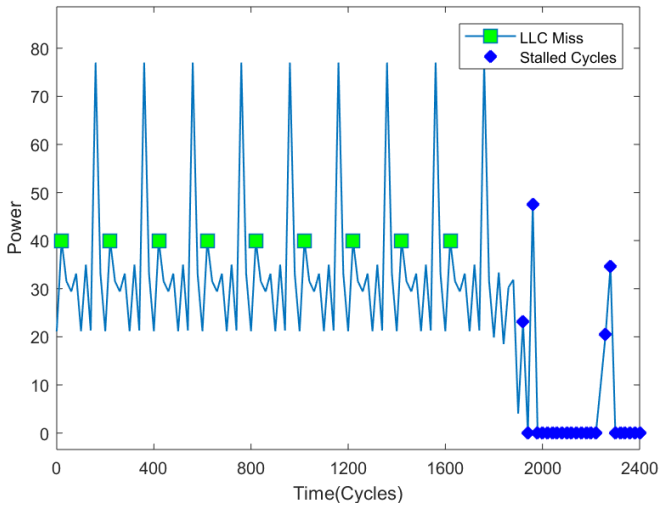


Figure 5.2: Special case of LLC miss: Processor doesn't stall for all LLC misses.

allows the processor to fetch more instructions, staying busy until the second LLC miss ends, etc. until eventually the processor encounters a miss during which it does run out of resources and is forced to stall. Since the first several LLC misses in this example do not cause any stall activity, a detector of LLC misses based on identifying the resulting stalls in the signal will fail to detect those misses, which will cause under-counting of LLC miss events. However, since those misses do not introduce stalls, their impact on performance is much lower compared to stall-inducing misses, so the signal-based reporting of miss-induced stalls in this example would still be close to the actual performance impact of these misses.

Another situation in which the LLC miss count would be under-reported involves overlapping LLC misses. One example of such a situation is shown in Figure 5.3, where an I$ access and a D$ access are both LLC misses that overlap and the processor is stalled during that overlapped time. When analyzing the side-channel signal to identify stalls caused by LLC misses, this usually results in reporting only one stall, which would be counted as one LLC miss. If the goal of the profiling is to actually count LLC misses, rather than account for their impact on performance, this will cause under-counting of LLC misses. However, this situation is a typical example where the MLP allows the performance penalties of mul-

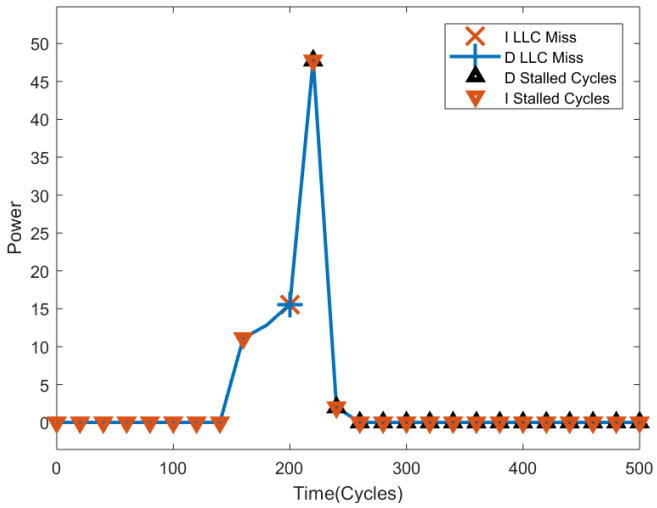


Figure 5.3: Special case of LLC miss: Overlapping LLC misses causes overlapping stalls.

tiple LLC misses to overlap with each other, and the resulting overall performance impact is not much worse than if there was only one LLC miss, so the signal-based reporting of how many long stalls occur and the performance impact of each such stall still very accurately tracks the actual performance impact of the LLC misses.

5.2.2 Memory Access in Physical EM Side-channel Signal

We now run the same small application on an A13-OLinuXino-MICRO [51], an IoT prototyping board. We use a near-field magnetic probe to measure the board's EM emanations centered around the processor's clock frequency. Even though we are using EM emanations from a real system, rather than power consumption from a simulated system, stalls produced by cache misses in this system still produce signal patterns (Figure 5.4) similar to that observed for the simulator - stalls cause a significant decline in signal magnitude. The stalls produced by most LLC misses lasts around 300 ns, with small variations that are likely due to variations in how much work the processor can do until it stalls after encountering an LLC miss.

However, the stalls caused by LLC misses exhibit several behaviors that were not encountered in the simulator because it used a simplified model of the main memory. One

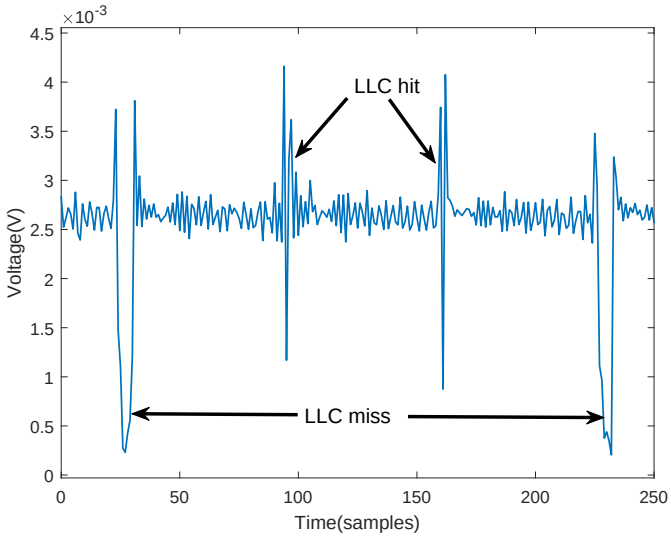


Figure 5.4: LLC hit and miss from physical side-channel signal of A13-OLinuXino-MICRO IoT device.

such behavior, shown in Figure 5.5, occurs when an LLC miss occurs while the memory is performing its periodic refresh activity.

We observed that a stall for an LLC miss that coincides with a memory refresh lasts approximately 2-3 μs, and this situation occurs approximately at least every 70 μs for the on-board H5TQ2G63BFR SDRAM chip [102]. Since these stalls do affect program performance and (especially) the tail latency of memory accesses, we count them (and account for their performance impact) separately when reporting our experimental results.

5.3 EMProf: Detection Algorithm

Because cache misses occur at a time when the processor is busy, the signal shape at the point of the miss changes significantly depending on the surrounding instructions and the processor's schedule for executing them. Therefore, rather than attempting to recognize the signal that corresponds to the misses themselves, which would require extensive training for each point in the code where the misses can occur, EMPROF detects the signal activity that corresponds to a stalled processor. The first step in this detection is to normalize the signal to compensate for the variation among different devices, measurement setup, etc. For

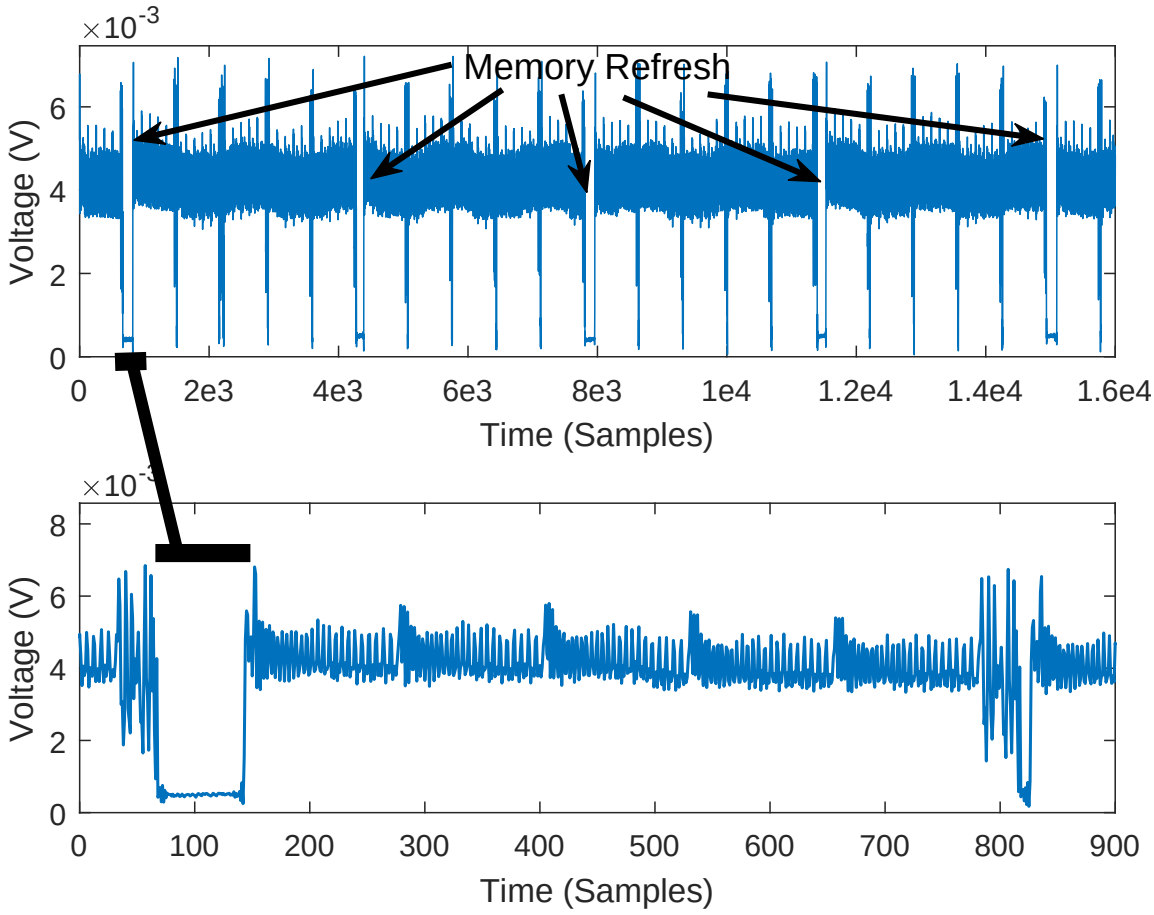


Figure 5.5: Memory refresh in A13-OLinuXino-MICRO IoT device. *(top)* shows memory refresh occurring in place of LLC miss, *(bottom)* shows a zoomed-in memory refresh.

example, we have found that even small changes in probe/antenna position can dramatically change the overall magnitude of the received signal. However, this change largely consists of a constant multiplicative factor that is applied to the entire signal. Similarly, the voltage provided by the profiled system's power supply vary over time. The impact of power supply variations on the EM emanations is largely that signal strength changes in magnitude over time. EMPROF compensates for these effects by tracking a moving minimum and maximum of the signal's magnitude and using them to normalize the signal's magnitude to a range between 0 (which corresponds to the moving minimum) and 1 (which corresponds to the moving maximum). EMPROF then identifies each significant *dip* in the signal whose duration exceeds a threshold. The threshold is selected to be significantly shorter than the

LLC latency but significantly longer than typical on-chip latencies.

One of the key advantages of EMPROF is that it can efficiently identify LLC-miss-induced processor stalls without any a-priori knowledge about or training on the specific program code that is being profiled. This enables EMPROF to profile execution equally well regardless of the program analysis tools and infrastructure available for the target system and software. One specific example of this versatility is that EMPROF can be used for memory profiling of the system's boot from its very beginning, even before the processor's performance monitoring features are initialized and before the software infrastructure for saving the profiling information is loaded.

5.4 Validation

EMPROF's goal is to identify in the side-channel's signal each LLC miss that stalls processor and to measure the duration of that stall. Since existing profiling methods do not operate at such level of detail, validation of EMPROF is a non-trivial problem. One problem is that hardware performance counters, such as `LLC-load-misses`, count all LLC miss events, including those that cause no stalls and those that overlap with other misses. Another problem is that their accuracy is reduced by either high interference with the profiled execution (when sampling the counter value often) or significant sampling error (when sampling the counter value rarely), so they are not very effective for relatively brief periods of execution. One illustration of this is that, as will be reported in more detail in Section 6.5, when using `perf` on A13-OLinuXino-MICRO to count LLC misses for a small application that was designed to generate only 1024 cache misses, the number of misses reported by `perf` had an average of 32,768 and a standard deviation of 14,543.

Thus we validate EMPROF's results in two ways. First, we engineer a microbenchmark that generates a desired number of LLC misses and compare EMPROF's reported LLC miss count to an a-priori-known number of misses. Second, we apply EMPROF to the power side-channel signal generated through cycle-accurate simulation, and then compare EM-

Table 5.1: Specifications of Experimental Devices

	Alcatel	**Samsung**	**Olimex**
Processor	QS[a] MSM8909	QS[a] MSM7625A	AW[b] A13 SoC
Frequency	1.1 GHz	800 MHz	1.008 GHz
#Cores	4	1	1
ARM Core	Cortex-A7	Cortex-A5	Cortex-A8

[a]Qualcomm Snapdragon
[b]Allwinner

PROF's results to the simulator-reported ground truth information about where the misses occur in the signal's timeline and where the resulting stall begins and ends.

5.4.1 Experimental Setup

To demonstrate EMPROF's ability to profile LLC misses, we ran the engineered microbenchmark on Android-based cell phones - Samsung Galaxy Centura SCH-S738C [103] and Alcatel Ideal [104], and on the Olimex A13-OLinuXino-MICRO [**olimex**] IoT prototype board. More detail on these three devices is provided in Table 5.1.

In our experimental setup, the signals received using a small magnetic probe are recorded using a Keysight N9020A MXA spectrum analyzer [105]. The spectrum analyzer was used for initial studies as it has built-in support to visualize signals in real-time, offers a range of measurement bandwidth and perform basic time-domain analysis.

5.4.2 Validation by Microbenchmarking

We implement a microbenchmark which generates a known pattern of memory references leading to LLC misses. The access pattern of the microbenchmark can be adjusted to produce LLC misses in groups, where the number of LLC misses in a group and the amount of non-miss activity between groups can be controlled. We refer to the total number of expected LLC misses as *TM* and the number of LLC misses consecutively in groups as *CM*. For example, if *TM*=100 and *CM*=10, the microbenchmark creates 10 groups of 10 consecutive LLC misses, each group separated by a micro-function call. Figure 5.6 provides a

```
1  // perform page touch
2  for(#pages_to_be_used)
3      load(page(cache_line_0))
4
5  // empty for loop
6  exec_blank_loop()
7
8  // perform memory loads
9  // TM: Total Misses requested
10 while(num_accesses != TM)
11     page = rand()
12     cache_line = rand()
13     addr = page*PAGE_SIZE + cache_line*CACHE_LINE_SIZE
14     load(addr)
15     // CM: Consecutive Misses occurring in groups
16     if(num_accesses % CM == 0)
17         micro_function_call()
18     num_accesses++
19
20 // empty for loop
21 exec_blank_loop()
```

Figure 5.6: Pseudo code of the microbenchmark.

brief overview of the microbenchmark code.

First, every page is accessed once to avoid encountering page faults later. Then the microbenchmark executes a tight `for` loop with no memory accesses. The signal that corresponds to this loop has a very stable signal pattern that can be easily recognized, which allows us to identify the point in the signal where this loop ends and the part of the application with LLC miss activity begins.

Next the microbenchmark executes a section of code that contains the LLC misses. The access pattern accesses cache-block-aligned array elements (so that each access is to a different cache block), with randomization designed to defeat any stride-based pre-fetching that may be present in the processor. Finally, another tight `for` loop allows us to easily identify the point in the signal at which the carefully engineered LLC miss activity has ended.

Figure 5.7 shows the overall EM signal from one execution of this microbenchmark on the A13-OLinuXino-MICRO board. The "memory accesses" portion of the signal includes many LLC misses, which occur in groups of 10, i.e. CM=10. A zoom-in that shows the signal for one such group of 10 LLC misses is also shown in Figure 5.7.

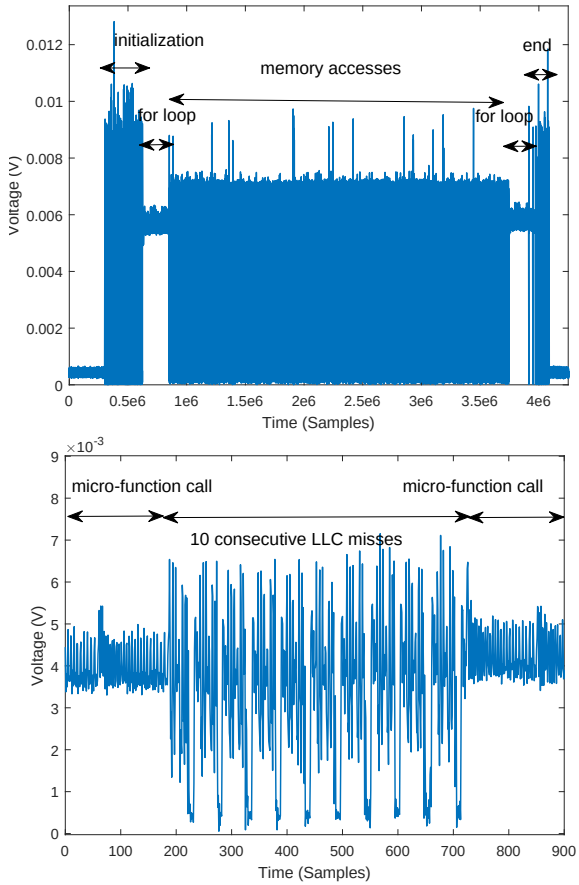


Figure 5.7: EM side-channel signal from microbenchmark on A13-OLinuXino-MICRO IoT device. *(top)* shows the entire run and *(bottom)* zooms into a section of LLC miss with CM=10.

Because the number of misses in the "memory accesses" section of the microbenchmark is known, and because this section can easily be isolated in the signal, we can apply EMPROF to this section and compare the LLC miss count it reports to the actual LLC miss count the section is known to produce. The results of this comparison were used to compute EMPROF's accuracy shown in Table 5.2. On all microbenchmarks the accuracy of EMPROF's LLC miss counting is above 99%, with an average of 99.52%.

5.4.3 Validation by SESC simulator

To evaluate EMPROF's ability to measure the duration of each LLC-miss-induced stall and to deal with overlapping LLC misses that occur in real applications, we use a sim-

Table 5.2: Accuracy of EMProf for microbenchmarks on Alcatel cell phone, Samsung cell phone and Olimex IoT device.

Benchmark		**Devices**		
#TM	**#CM**	**Alcatel**	**Samsung**	**Olimex**
256	**1**	99.61%	99.22%	99.61%
256	**5**	100%	99.61%	99.22%
1024	**10**	99.41%	99.61%	99.51%
4096	**50**	99.83%	99.71%	98.98%

ulator configuration that mimics the processor and cache architecture of the Olimex A13 OLinuXino-MICRO board. The simulator is enhanced to produce a power consumption trace that will be used as a side-channel signal in EMPROF, and also to produce a trace of when (in which cycle) each LLC miss is detected and when the resulting stall (if there is a stall) begins and ends.

We first run the same microbenchmarks we ran on the Olimex board and compare the two signals (Figure 5.8). We observe that, although one signal is produced by the simulator's (unit-level) accounting of energy consumption and the other is produced by actually receiving EM emanations from a real processor, the relevant aspects of the two signals are similar in nature - the loops used to mark the beginning and ending of the "memory accesses" section in the microbenchmark are clearly visible and can be easily identified in the signal, and the LLC misses themselves exhibit similar behavior in the signal. The most prominent difference between the two overall signals is that the start-up and tear-down involves much more activity on the real system than on the simulator, primarily because the simulation begins at the entry point and ends at the exit point of the microbenchmark's executable, whereas in the real system the start-up and tear-down also involves system activity, e.g. to load and prepare the executable for execution before any of the executable's instructions are executed.

Having determined that the simulator's power signal is a reasonable proxy of the real system's EM signal for the purposes of EMPROF validation, we proceed to run the microbenchmarks and several SPEC CPU2000 benchmarks in the simulator, analyze the re-

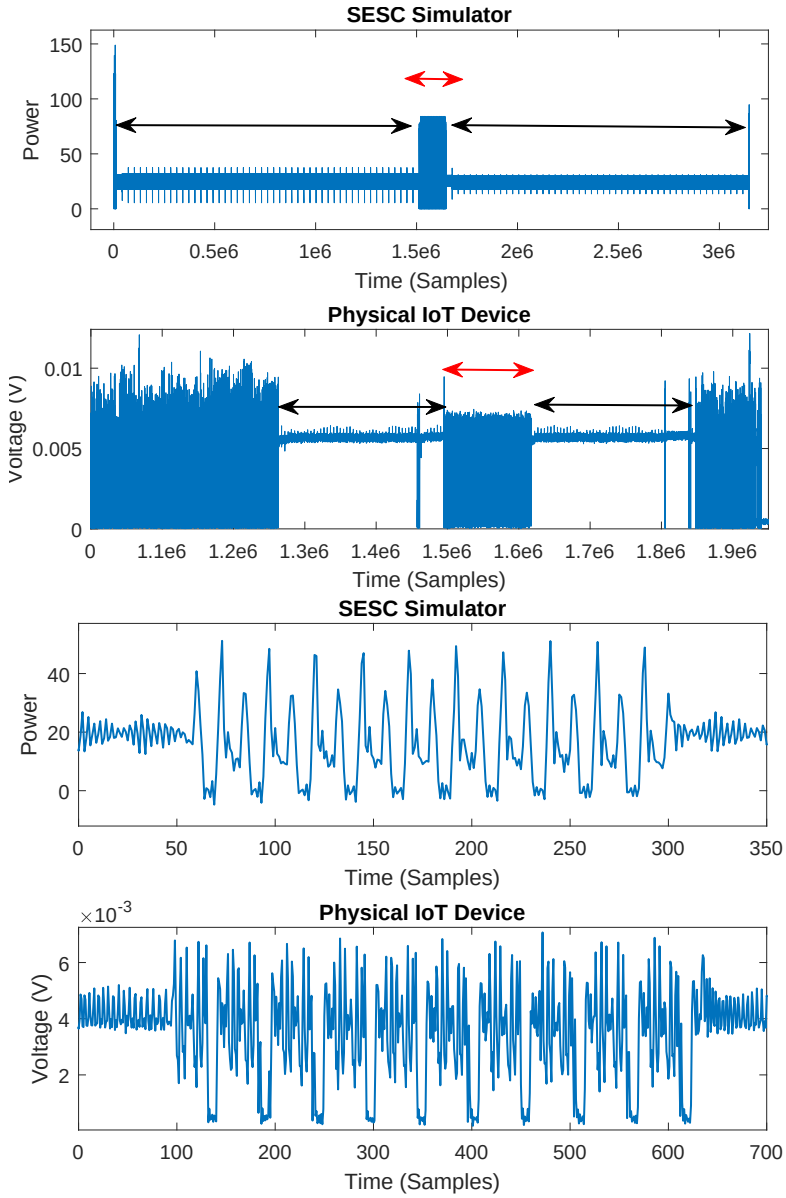


Figure 5.8: Microbenchmark run on SESC simulator and Olimex A13-OLinuXino-MICRO IoT device. *(top)* compares the entire microbenchmark runs, with red bold arrows pointing to the segment of memory accesses and black dashed arrows pointing to the empty `for` loops. *(bottom)* is a zoomed-in segment of a consecutive LLC misses with *CM*=10.

sulting power signals in EMPROF, and compare EMPROF's results to the ground-truth results recorded by the simulator. The results of this comparison are shown in Table 5.3 in terms of accuracy of the reported LLC miss count and the reported LLC-miss-induced stall cycles.

Table 5.3: Accuracy of EMProf on simulator data for benchmarks.

Benchmark		Miss Accuracy(%)	Stall Accuracy(%)
#TM	**#CM**	**Microbenchmark**	
256	**1**	97.7	99.3
256	**5**	97.8	99.3
1024	**10**	99.4	99.9
4096	**50**	99.8	99.8
SPEC CPU2000			
ammp		99.67	98.4
bzip2		95.2	99.96
crafty		99.31	100
equake		93.2	98.5
gzip		99.96	99.8
mcf		99.9	99.5
parser		99.9	99.7
twolf		99.16	100
vortex		99.04	99.9
vpr		100	99.24

5.4.4 Validation by EM Side-channel Signal from Main Memory

Finally, we wanted to systematically verify that the stalls reported by EMPROF coincide with actual memory activity. Intuitively, when a memory request is getting fulfilled by the main memory, the activity level in main memory should increase. Similar to how we receive the EM emanations from the processor, we can also receive the EM emanations from main memory. By receiving the two signals simultaneously and checking whether the dips in the processor's signal coincide with memory activity, we can gain additional confidence that the dips detected by EMPROF are indeed a consequence of LLC misses.

Unfortunately, not every device is emenable to such dual-probing - the processor and memory tend to be physically close to each other, so the physical placement of both probes is a challenge. However, we found that on the Olimex board, the processor and memory are relatively spaced apart, allowing simultaneous probing with no interference. We additionally use a passive probe to measure the CAS pin activity off a resistor, and this experimental setup is shown in Figure 5.9.

Table 5.4: Statistics of total LLC misses and total percentage latency in execution time obtained from EMProf for Alcatel cell phone, Samsung cell phone and Olimex IoT device.

Benchmark		Total LLC Misses			Miss Latency (%Total Time)		
		Devices			Devices		
		Alcatel	Samsung	Olimex	Alcatel	Samsung	Olimex
#TM	#CM	Microbenchmark					
256	**1**	257	254	255	0.92	3.57	9.44
256	**5**	256	255	258	1.15	4.06	10.10
1024	**10**	1030	1020	1029	1.00	4.19	9.88
4096	**50**	4103	4084	4138	2.05	4.55	10.25
SPEC CPU2000							
ammp		20511	280141	174142	2.42	7.59	9.06
bzip2		451103	3353123	5932148	2.15	1.04	6.59
crafty		314385	901153	675781	2.44	0.38	1.52
equake		627734	2138132	5925404	2.68	0.61	7.49
gzip		152264	1001392	513256	1.11	0.39	1.21
mcf		303365	895295	546714	5.22	7.18	3.28
parser		365412	1934334	2318384	2.19	3.39	8.63
twolf		35086	1014888	228465	1.03	4.84	3.00
vortex		235667	897317	533784	3.63	2.03	2.9
vpr		4970	100750	203130	0.09	0.23	0.6
Average		251049.7	1251652.5	1705120.8	2.3	2.77	4.43

Figure 5.10 shows the magnitude of the processor's and the memory's side-channel signal for CM=10 to the main memory that are separated by non-memory activity. We observe that an LLC miss causes the processor's signal magnitude to drop significantly, while the memory request caused by the LLC miss result in a sudden burst of memory activity.

Finally, one may intuitively expect that the memory's EM signal might be a better indicator of LLC misses than the processor's signal. Unfortunately, this is not true - while

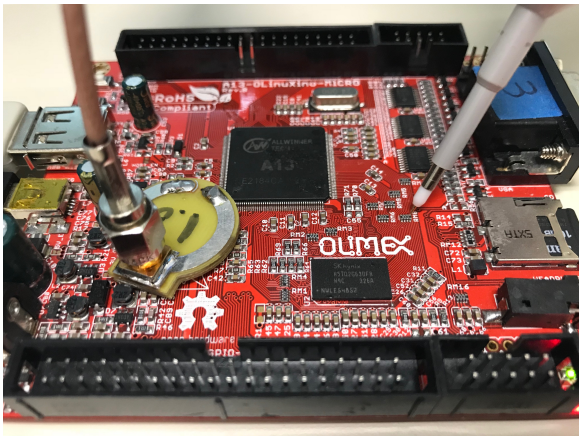


Figure 5.9: Olimex board measurement setup for dual-channel probing of processor and memory signals simultaneously.

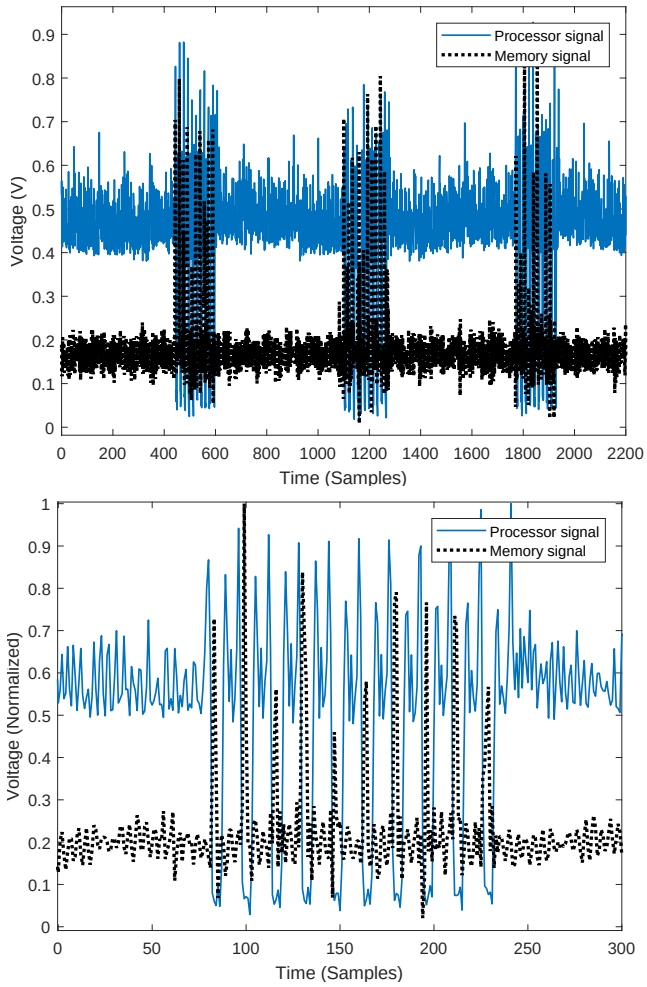


Figure 5.10: EM side-channel emanations of Olimex board from both processor and memory side for microbenchmark with *CM*=10. *(top)* shows three groups of LLC misses with micro-function call in between, *(bottom)* shows a single zoomed-in group.

it is true that memory activity occurs on each LLC miss, memory activity also occurs due to DMA transfers, DRAM refreshes, and many other reasons that are completely unrelated to LLC misses. Furthermore, by measuring stalls in the processor's EM signal, we obtain information that is more relevant for performance optimizations - how often the processor stalls due to LLC misses and how long these stalls last.

5.5 Results

To evaluate the effectiveness of EMPROF on real-world applications, it was additionally applied to ten SPEC CPU2000 benchmarks [106] on the two Android cellphones and one IoT device. The integer SPEC CPU2000 benchmarks were chosen as they display a range of realistic memory behaviors [58].

The final results of SPEC CPU2000 benchmarks were recorded using near-field magnetic probe, by first down-converting the signal using a ThinkRF Real Time Spectrum Analyzer WSA5000 [107] and then digitizing the signal using a computer fitted with two PX14400 high speed digitizer boards from Signatec [108]. This was needed because the N9020A MXA has a limit on how long it can continuously record a signal, and most SPEC benchmarks execution times significantly exceed this limit.

5.5.1 Profiling Results

Table 5.4 shows the results for the total number of LLC cache misses and the number of stall cycles (as a percentage of the benchmark's execution time), as reported by EMPROF for each benchmark on each target device.

The number of LLC misses is much lower for the Alcatel cellphone than for the other two devices, mainly because the LLC in Alcatel is 1 MB while Olimex and Samsung device both have a 256 KB LLC, so the LLC miss rate for Alcatel can be expected to be much lower. Samsung device's processor has a hardware prefetcher, so it is able to avoid some of the LLC misses that occur in the Olimex device, and therefore the number of misses Olimex can be expected to be higher than in the Samsung phone even though their LLCs have the same size. Additionally, the processor in the Olimex board has a higher clock frequency than the processor in the Samsung phone, while their main memory latencies (in nanoseconds) are very similar, creating more stall time per miss in the Olimex board and also allowing fewer LLC misses to be completely hidden by overlapping them with useful

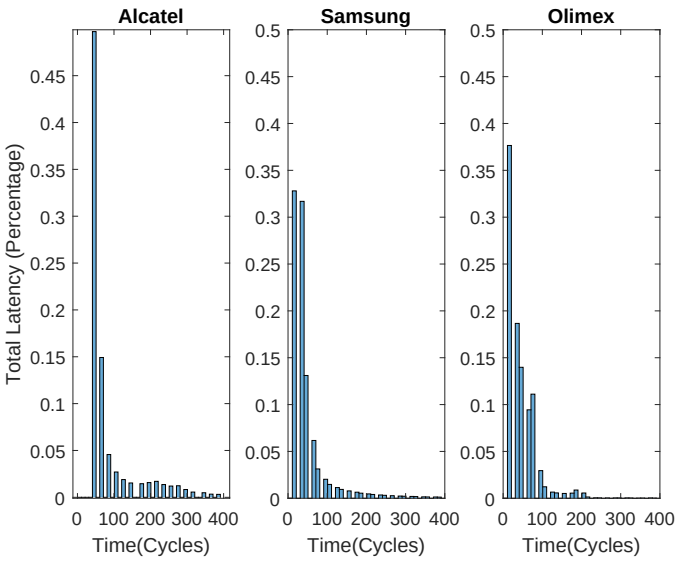


Figure 5.11: Histogram of stall latencies obtained for SPEC CPU2000 *mcf* benchmark for Olimex IoT device, Alcatel cell phone and Samsung cell phone.

work in the processor.

While the total stall cycles provides an indication of how performance is affected by LLC misses overall, a key benefit of EMPROF is that it also provides information about the stall time of each LLC miss. Figure 5.11 shows the histogram of LLC miss latencies observed on the three devices for the SPEC CPU2000 benchmark *mcf*. Most stalls are brief in duration, mostly due to the processor's ability to keep busy as the miss is being serviced. However, a significant number of stalls last hundreds of cycles, and we observe that, compared to the IoT board, the two phones have a thicker "tail" in the stall time histogram.

5.5.2 Effect of Varying Measurement Bandwidth

During our initial study, we tested the effect of a varying measurement bandwidth from 20 MHz, 40 MHz, 60 MHz, 80 MHz and 160 MHz. Low measurement bandwidth may not introduce enough samples to exemplify the cache miss feature, and with increasing measurement bandwidth, an increase in sampling rate correlates to a better and more accurate detection of LLC miss and its associated latencies. Figure 5.12 illustrates this effect

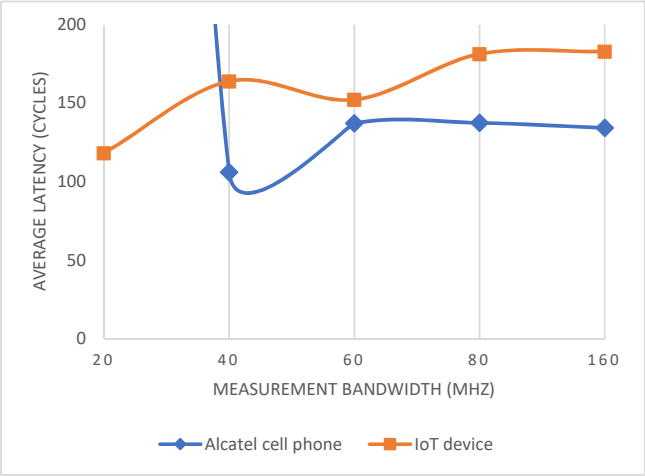


Figure 5.12: Effect of varying measurement bandwidth for SPEC CPU2000 *mcf* benchmark across Alcatel cell phone and IoT device.

for Alcatel cell phone and A13-OLinuXino-MICRO running SPEC CPU2000 *mcf* benchmark. In the IoT device, a lower sampling rate mostly results in less accurate stall latency determination. However, for the Alcatel cell phone the lowest sampling rates also prevent detection of many LLC-induced stalls, such that at 20 MHz EMPROF detects only the very few stalls that have extremely long durations (their average duration is 1100 clock cycles). For both devices, the average stall time stabilizes at 60 MHz or more of measurement bandwidth, indicating that bandwidth equivalent to only 6% of the processor's clock frequency is sufficient to allow identification of LLC-miss-induced stalls in the signal.

5.5.3 Profiling Boot Sequence

One of the most promising aspects of EMPROF is its ability to profile hard-to-profile runs, such as the boot sequence of the device. Figure 5.13 shows the rate of total LLC misses as time progresses for two boot-ups of the IoT device. These results can be used to, for example, decide whether memory locality optimization should be considered as a way to speed up the boot of the device.

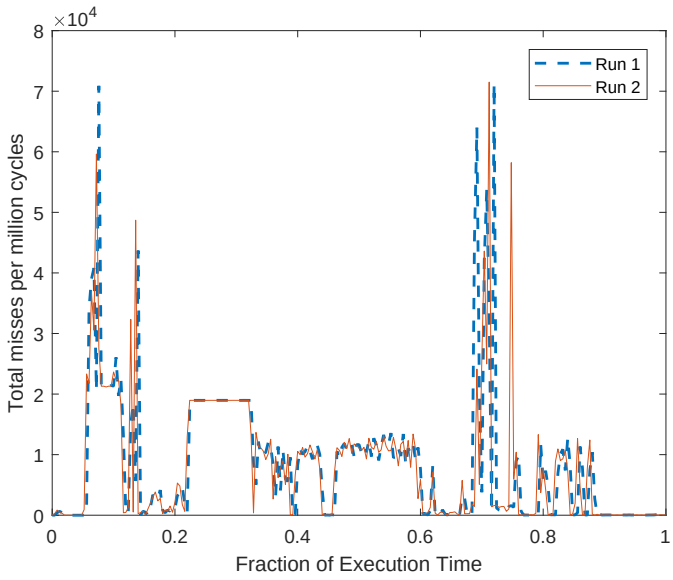


Figure 5.13: Boot sequence EMPROF profiling for two distinct runs on IoT device.

5.5.4 Code Attribution

While accurately profiling stalls due to LLC misses in execution time provides very valuable insight into performance, it would be even more helpful for developers to be able to identify in which parts of the code these LLC misses and stalls are happening. Ideally, such attribution of misses to code would also be based on the EM signal, to retain the zero-interference advantages of EMPROF. Several methods that attribute parts of the signal to application code have been reported in the literature, including Spectral Profiling [22], EDDIE [109], which can attribute parts of the signal to the code at the granularity of loops, and even ZOP [17], which can achieve fine-grain attribution of signal time to code albeit that requires much more computation so it may not be feasible for long stretches of execution. By applying one such scheme and EMPROF to the same signal, the LLC miss stall discovered in the signal by EMPROF's could be attributed to the application code in which they occur. To illustrate this, Figure 5.14 shows how the spectrum (horizontal axis) for the SPEC CPU2000 benchmark *parser* changes over time (vertical axis). There are three distinct regions that can be observed in this spectrogram, which correspond to three functions in *parser*. To illustrate how signal-based attribution would be used in conjunction

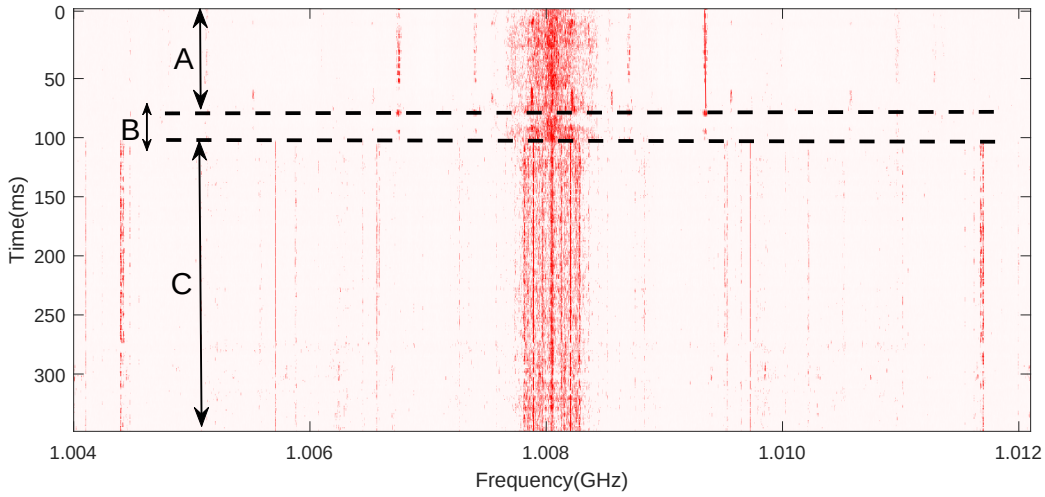


Figure 5.14: Spectrogram of SPEC CPU2000 *parser* benchmark

with EMPROF, we (manually) mark the transitions between these function in the signal as shown by horizontal dashed lines in Figure 5.14 and attribute misses in each part of the signal to the corresponding function. Table 5.5 shows EMPROF's results with this attribution. As we see, the batch_process function should be the main target for optimizations that target LLC misses – it occupies the largest fraction of execution time, it suffers the highest LLC miss rate, and it has the highest fraction of its execution time spent on stalls caused by these LLC misses. We note that this only serves to illustrate how this attribution would work - finer-grain analysis (à la Spectral Profiling) of the signal would identify multiple loops in some of these functions and thus likely provide more precise (and informative) results than those shown in Table 5.5.

Table 5.5: Observable loops in SPEC CPU2000 *parser* benchmark.

Region	**Function**	**LLC Miss Rate**	**Mem Stall Cycles (per Million Cycles)**	**Avg. Miss Latency**(%)
A	read_dictionary	2667.71	1.63	218.71
B	init_randtable	317.66	0.18	211.85
C	batch_process	16795.47	10.17	217.72

5.6 Conclusions

This paper presents EMPROF, a new approach to profile memory behavior in resource-constrained systems such as IoT and hand-held devices. EMPROF does not require any hardware or software support including hardware counters or instrumentation in target machine. EMPROF measures the EM emanated signal from the processor without any support or assistance of the device-under-test. Hence, it is totally *observer-effect free* and has no interference with the running application such as memory pollution or frequent interrupts. It utilizes this fact that signal level drops when processor gets stalled. By dynamically detecting this signal level, EMPROF can accurately find main memory accesses which cause stall in processor and matter for the purpose of performance profiling very accurately in execution time. It is also able to measure effective latency associated with each of these misses.

To validate EMPROF, microbenchmarks with known memory behavior are used. Also, EMPROF has been tested on simulated side-channel signal where EMPROF is able to pinpoint expected cache misses with 99.1% accuracy in microbenchmarks and 98.1% in for real applications. Two Android cellphones and an IoT device have been used to evaluate EMPROF. It shows a cumulative 99.5% accuracy on acquired signal from these devices. We illustrated how EMPROF can profile execution even where no other profiler can be used, such as the boot sequence in these devices. Moreover EMPROF can be used with other profilers which are able to profile execution and attribute EM emanation to code sections.

CHAPTER 6
BLIND SOURCE SEPARATION OF ELECTROMAGNETIC SIDE-CHANNEL SIGNALS

6.1 Overview

Previous chapters have shown promising results that EM side-channel signals are potentially applicable for various purposes. However, there are *two* major issues that limit practicality of such methods. First, while IoT and embedded devices tend to use simpler processors, the use of multi-core processors and SoC chips is rapidly increasing. In such systems, more than one computing unit is active on the chip, and more than one application is executing, thus execution monitoring of a specific processing unit (e.g., a core) suffers from "interference" signals from other active processing units on the same chip. Second, one of the greatest benefits of using EM emanations is that they can be received at a distance [110]. However, as IoT devices proliferate, signals from multiple devices, possibly even multiple devices of the same design, are *superimposed* together when they are received by the monitor's antenna, thus creating another scenario where monitoring of any specific device suffers from "interference" signals emanated by others.

In this chapter we demonstrate the feasibility of decomposing the acquired superimposed signals into per-source components in both multi-core and multi-device scenarios. We observe that this problem is closely related to the well-known *Cocktail Party Problem* [111], and explore two methods that recover per-source signals without any priori information about the sources or the "signal-mixing" process. This separation block can be used as a preprocessing block. Figure 6.1 suggests how the previous studies can benefit from source separation. We envision using this method as a preprocessing block, to operate in a multi-core or multi-device environment, which enables numerous previously proposed

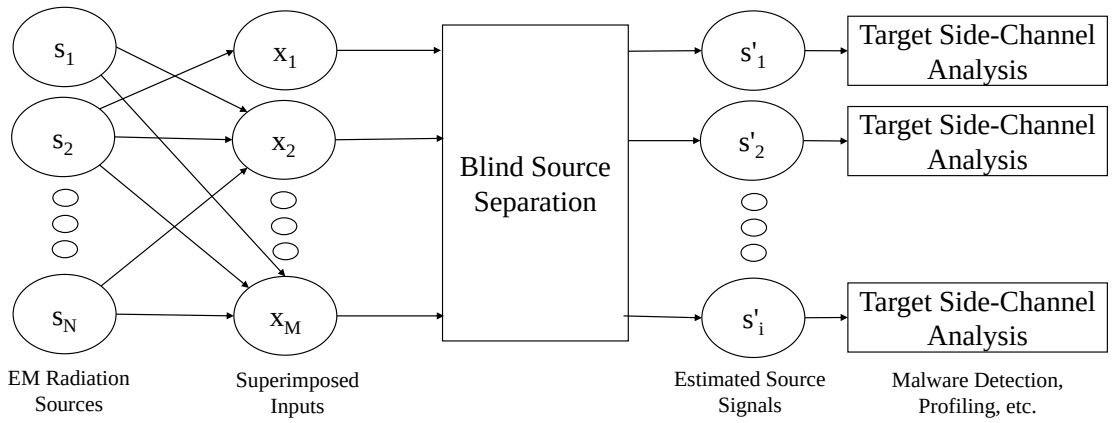


Figure 6.1: The envisioned use of source separation block. Each input channel, x_i, receives a superposition of multiple EM emanations sources, s_i. The blind-source separation (BSS) block estimates per-source signals, s'. Previously proposed methods for profiling, malware detection, etc. can the be directly applied on its outputs.

studies of EM side-channel signal that operate based on the frequency-domain, e.g. Spectral Profiling, EDDIE, or work based on an time-domain analysis of a received signal (e.g., template matching, etc.) [113, 114, 115, 116, 117].

To achieve this goal, first, we explore the use of the most common approach in blind source separation, *Independent Component Analysis (ICA)*. Although ICA methods can enhance the quality of the signals by increasing the signal-to-interference-plus-noise (SINR), they are unable to dramatically reduce the interference and provide a signal for usage in malware detection, profiling or cryptographic analysis. Hence, we propose to take advantage of the existing disparity in emanation of EM side-channel from various on-chip sources. We use a time-frequency masking method based on DUET [118] to recover source signals. We show that this method is suitable and effective due to unique characteristics of processor side-channel signals and software applications' time-frequency footmarks. We implement a proof-of-concept system that uses previously proposed time-frequency masking in its core to separate signals generated by on-chip cores. We also show this method is effective in separating emanated signals from adjacent IoT devices. To show the benefit of this method, we implement a simple proof-of-concept malware detector that gets trained

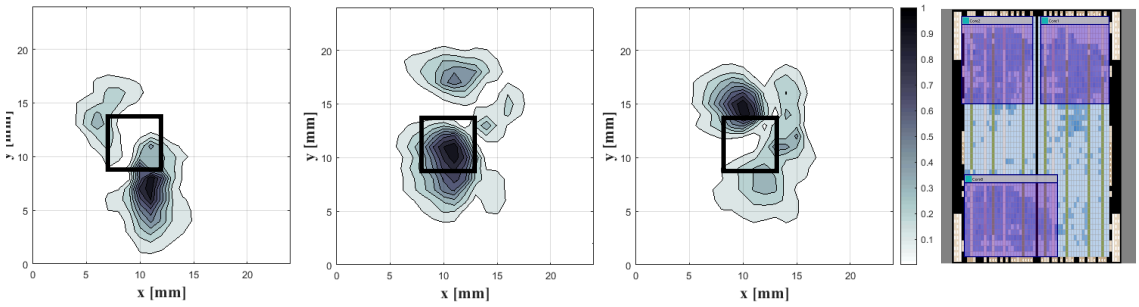


Figure 6.2: Three heat-maps of clock-modulated EM emanations of 3 on-chip cores on the left and their corresponding floor plan on the right. The black box shows the location of the ASIC chip.

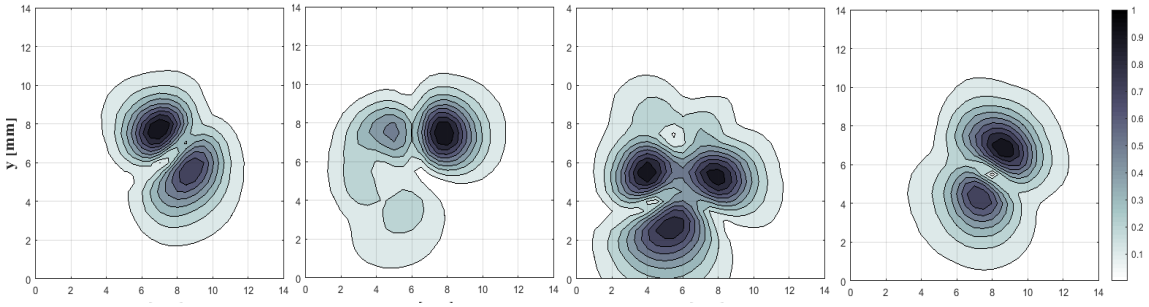


Figure 6.3: Four Heat-maps of clock-modulated EM emanations of core one to four (left to right) of A33 SoC.

on a single source signal to monitor an application. Then it exploits the source separation block in its front-end to monitor the application on a multi-core processor without any change or retraining.

The rest of this chapter is as follows, in Section 6.2 we present experiments on localization of on-chip sources, in Section 6.3 a background on blind source separation is discussed. In Section 6.4 the separation method is explained. In Section 6.5 experimental results on an IoT and an FPGA device are presented. Lastly, Section 6.6 concludes this chapter.

6.2 Localization of leakage sources

In CMOS technology, state transition in inverters and coupling due to the extreme proximity of components (e.g., transistors) in the device can cause unintentional changes in electromagnetic field [119, 120] and hence an EM-based side-channel signal. Previous work has shown that the clock signal acts as a carrier signal and these emanations manifest themselves as information-rich modulations of it [120, 22].

Although EM measurement is usually conducted by a near-field probe located in close distance of the processor, there is still a substantial interference from nearby components on the device. Hence, strengthening the received signal from target source (i.e., increasing the signal-to-noise ratio) is one of the main challenges in such measurements. Clearly, when more than one processing unit is active in relatively close proximity, more mutual interference is introduced to the system. This issue becomes even more challenging once two processing units are integrated on the same chip and share resources (i.e., multi-core CPUs).

Our observations show that the AM-modulated emanated signals spatially differ from one on-chip core to another. Thus, we can leverage this spatial variation not only for the localization of strongest sources but to extract per-core signals from superimposed recordings. When there are multiple sources of leakage, sources corresponding to a core can be physically close to each other and far enough from the other cores to be considered as one physical source of EM emanations.

To further study this, we implement three NIOS®-II cores on an Altera Cyclone V FPGA using a Terrasic DE0-CV development board. Figure 6.2 shows the chip plan of this implementation. The FPGA chip package is $22mm \times 22mm$ and the de-packaged silicon die is about $6mm \times 6mm$. A microbenchamrk consisting a simple loop with 10 instructions in its body is executed on each core. Each core operates at $50MHz$ and the loop generates a clock-modulated peak on $51.2MHz$. The amplitude of this frequency component is

measured by a measurement probe that sweeps over the chip by a XY-Plotter with 0.5mm steps. The resulting heat-map is presented in figure 6.2. The heatmaps resembles the chip plan, however, we cannot confirm whether the sources of leakage are on the silicon die itself or the surrounding connections inside the package. These three cores are integrated in a very close proximity. However, the AM-modulated signal from each core is yet received with different attenuations in each spatial point. We repeated this experiment when all the three cores simultaneously run three microbenchmarks. Each benchmark produced a spike in a different frequency component. The new heatmaps also resemble the ones presented in Figure 6.2.

We replicate this experiment on an off-the-shelf IoT device (A33-OLinuXino) with 4 on-chip ARM cores. In each repetition we run the benchmark on a different core and use Linux *task affinity* command to prevent it from changing to another core. Same as the previous observation, emanation patterns vary when the active core changes. Figure 6.3 shows the resulting heatmaps. Since we do not have an access to the chip plan of this SoC, we are not able to specify the physical component that generates the EM leakage (e.g., pin, power management unit, etc.). However, the change in emanation patterns hints that there are different physical sources of emanations for each of these cores. The observed disparity in leakage patterns provides an opportunity to separate EM signals generated by each of these cores.

6.3 Cocktail Party Problem

The *cocktail party* phenomena simply refers to brain's ability to focus on one discussion and treat any other chat as a background noise in a crowded party [111]. This phenomena has motivated decades long efforts for finding solutions to identify a set of independent source signals from a set of mixed observations without prior knowledge about mixing channel, shortly known as *blind source separation (BSS)*. Principally, BSS problem has been raised in various fields with different constraints, such as, to increase capacity in

wireless receiving models (MIMO systems), shock and vibration analysis in mechanics, to separate brain from muscle activity in electroencephalography (EEG) signals, etc.

The basic linear instantaneous BSS model is expressed as a matrix factorization and described as follows,

$$x(t) = As(t) + n(t), \tag{6.1}$$

where $x(t) \in \mathbb{C}^M$ is the observation vector, $s(t) \in \mathbb{C}^N$ is the source component vector, $n(t) \in \mathbb{C}^M$ is the noise vector, $A(t) \in \mathbb{C}^{M \times N}$ is mixing matrix, and the number of sources and observations implies $M \geqslant N \geqslant 2$. The system can have three scenarios, *determined* ($M = N$), *underdetermined* ($M < N$), and *overdetermined* ($N < M$). In a more sophisticated model, the mixing matrix, A, may incorporate the traveling delay along with relative attenuation by $A_{ij} = a_{ij}e^{-D_{ij}}$ where D_{ij} is due to traveling time delay. In the context of this paper, M is the number of recording channels(i.e. antennas) and N is the number of target EM emanations sources,i.g. cores. In this section, we use *mixing* interchangeably with *superposition*.

Assuming no apriori knowledge about mixing matrix and unknown source observations, the objective is to estimate a mixing matrix that produces source observations which meet a defined criteria. An assumptions on the statistical properties of the sources usually provide a basis for the de-mixing algorithm. The large number of developed solutions can be categorized into *four* classes based on the source separation condition or the restricted source features: independent component analysis (ICA), sparse component analysis (SCA), non-negative matrix factorization (NMF), and bounded component analysis (BCA). NMF and BCA solutions are out of the scope of this paper.

Classic ICA solutions assume the sources are non-Gaussian signals, and they are statistically independent from each other. Generally, they start with a pre-whitening stage. Then, to estimate source signals, they form a statistical cost function. The cost functions of ICA algorithms are always constructed according to different metrics of statistical independence, including maximum likelihood, mutual information, convex divergence, Kullback-

Leibler divergence, etc. They start with a random guess and use a gradient-descent optimization to find mixing matrix that minimizes the cost function. In ICA, the mixing matrix, A, is a square which means it only works for determined scenarios. While some of the ICA solutions perform well in the presence of Gaussian additive noise, any other noise or interference should be modeled explicitly as an independent source.

The ICA's assumptions , non-Gaussian distribution and independence of sources, may hold true for processor's EM side-channel signals. We evaluated the performance of Infomax [121, 122], JADE [123], FastICA [124], and M&S algorithms [125] for blind source separation on a multi-core device. We recorded two superimposed signals when two on-chip cores were executing two benchmarks. While these methods could increase the signal-to-noise ratio (SNR) of input signals, none of them could significantly reduce the SINR. The overall performance of ICA was not satisfactory. We can mention three reasons for this. First, ICA needs an input channel for each source. However, there are multiple sources of interference on the chip and device. Also, the measurements have high Gaussian noise and the performance of ICA methods are sensitive to input SNR. Second, most of the ICA methods perform well in instantaneous and fixed-delay models. Since the radiation is from a surface, the near-field probe receives the signal with slightly variable delays. Third, some sources of EM emanations are shared between both cores. This can cause some partial correlation between two signals.

BCA methods are proposed to address restriction of ICA method to accomplish separation of undetermined receiving and Gaussian mixtures. SCA relays on the sparseness of source signals as substitute of independence used in ICA. The sparseness of source signal means that $s(t)$ contains as many zeros as possible in time domain. Since not many types of signals are inherently sparse in original domain (e.g., mechanical vibration), we can seek enough sparseness in other transformed domains through specific transformation methods. Time-Frequency and wavelet transformation are often utilized for this purpose. Bofill *et al.* [126] exploited partial sparsity of speech signal in time-frequency. Jourjine *et al.* [118]

made an additional assumption that speech source signals are w-*disjoint orthogonal* and exploited that to estimate relative delay and attenuation across channels to separate speech mixtures. We suggest using a similar technique, with some modifications, for separation of superimposed EM emanations signals.

Two signals are ω-disjoint if their time-frequency representation does not overlap. Formally, any two sources, $s_i(t)$ and $s_j(t)$, are ω-disjoint if the supports of their windowed Fourier transform are disjoint, meaning:

$$\hat{s}_i \times \hat{s}_j = 0, \forall \omega, \tau, \tag{6.2}$$

where the windowed Fourier transform of $s_i(t)$ is defined as:

$$\hat{s}(\omega, \tau) = \int_{\infty}^{-\infty} W(t-\tau) s_i(t) e^{-j\omega t} dt, \tag{6.3}$$

and $W(t)$ is the window function.

Prior work [127, 128, 129] showed while speech or music source signals may not be fully ω-disjoint even in a high-resolution time-frequency space, the assumption can still achieve satisfactory results. If sources are imperfectly ω-disjoint, it is fair to assume each time-frequency bin is dominated by only one source. Hence, an ideal binary time-frequency mask, $M(\omega, \tau)$ can be used to de-mix the input signals, where I_j is interference from any other source in a (ω, τ) bin:

$$M_i(\omega, \tau) = \begin{cases} 1, & \text{if } (s_i(\omega, \tau) - I_j(\omega, \tau)) > treshold \ \ \forall j \\ 0, & \text{otherwise} \end{cases} \tag{6.4}$$

and the source signals can be presented as

$$\hat{s}_i(\omega, \tau) = M_i(\omega, \tau) \times \hat{x}(\omega, \tau). \tag{6.5}$$

Previous studies explored various methods of constructing $M(t, f)$ [118, 130, 131, 132, 133, 134, 135]. Generally, each method has two stages. First a set of features compares T-F bins in one channel to another in a bin-wise manner. Next, a clustering algorithm is used to classify T-F bins into classes which represent each source. We suggest that using a binary T-F masking method can be effective for source separation from superimposed EM emanation side-channel signals. Also we are able to cluster the T-F bins because of disparity between leakage patterns.

6.4 Blind Separation By T-F Masking

6.4.1 Characteristics of clock-modulated EM side-channel signal

By looking on results from Sectral profiling and also observing radiation signals when a set of benchmarks from MiBench [99] are running (see section 6.5.4), we conclude that we can make two assumptions about this type of EM signals. First, EM emanation of different software applications are sufficiently sparse in time-frequency domain. Second, they are w-disjoint orthogonal to each other. The assumption of sparsity in time-frequency domain is valid since the critical information is mostly carried by a relatively small subset of frequency components. Recent studies confirm this for many IoT benchmarks since they could achieve almost perfect representation of short-time spectra by only one or just a few components of it. Rickard *et al.* [129] provided a metric to evaluate how much two signals are w-disjoint orthogonal given a T-F mask (i.e., $M(\omega, \tau)$ from equation 6.5). We can measure the orthogonality of two signals by measuring the normalized deference between the signal's energy maintained in masking, s_i, and the interference energy maintained in masking, s_j. d_j is normalized between 0 and 1 (fully ω-disjoint orthogonal) and defined as

$$d_j(M) = 2^{D_j(M)-1}, \tag{6.6}$$

where

$$d_j(M) = \frac{\iint |M(\omega,\tau)\hat{s}_i|^2 d\omega d\tau - \iint |M(\omega,\tau)\hat{s}_j|^2 d\omega d\tau}{\iint |\hat{s}_j|^2 d\omega d\tau}. \tag{6.7}$$

In section 6.5.4 we compute the this metric, given an ideal mask for few benchmark, mixes to quantifiably show masking can effectively preserve characteristics of the applications.

6.4.2 Separation Algorithm

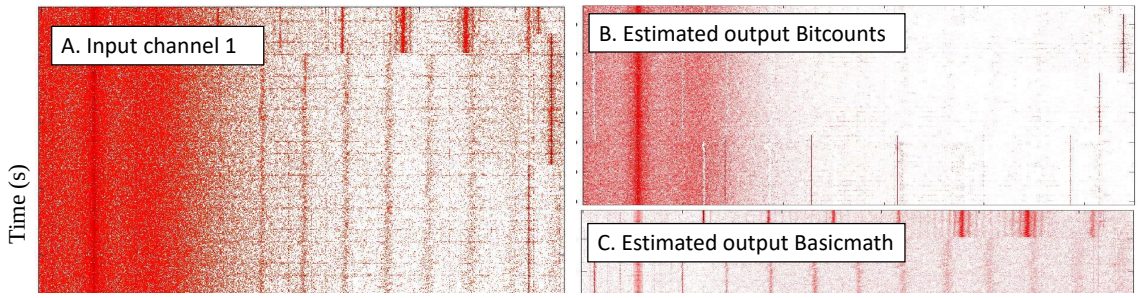


Figure 6.4: The outputs of the separation method on execution of Basicmath and Bitcount benchmarks running on an ARM (Olinuxino) board. Input channel one (on the left) is after pre-whitening and thresholding. The outputs of the separation algorithm are on the right.

Our separation algorithm uses a T-F masking method as outlined in 6.3. We particularly follow the comparison and classification that Rickard *et al.* [118] used for separation of speech signals. With some preprocessing and considerations we can use apply a similar technique on EM side-channel signals. The flow graph of this process is shown in Figure 6.6. We place two antennas above the processor chip. The EM signal is acquired by two channels while an applications is running on each core. A common prewhitening stage is applied to these channels. For each channel, the time-frequency representation is calculated by using short-time Fourier transform shown in (6.3). Then the DC component is excluded. By simple thresholding, we exclude insignificant T-F bins with less than 0.1% of window's energy from our analysis. Also we exclude all the T-F bins associated to the clock frequencies of each core. We experienced that these T-F bins add confusion to rest of

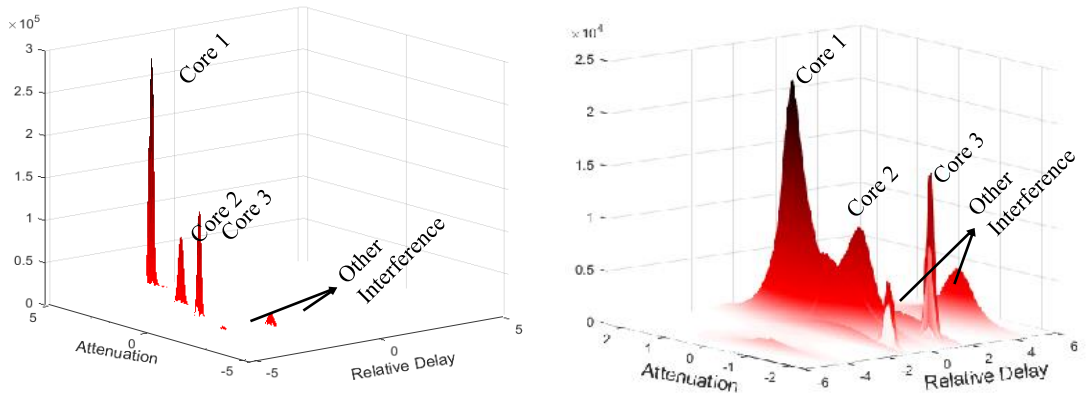


Figure 6.5: Histogram of relative delay and attenuation of three active cores on an FPGA(left) and A33-OLinuXino device(right). Each core is running a microbenchmark.

the process. We will add back these components to all outputs at the end.

To compare provide a bin-wise comparison, we calculate relative attenuation, a, and relative delay, σ, for each bin as:

$$a(\omega,\tau) = |\hat{x}_2(\omega,\tau)/\hat{x}_1(\omega,\tau)| - |\hat{x}_1(\omega,\tau)/\hat{x}_2(\omega,\tau)| \tag{6.8}$$

$$\sigma(\omega,\tau) = (-1/\omega)\angle(\hat{x}_2(\omega,\tau)/\hat{x}_1(\omega,\tau)) \tag{6.9}$$

We construct a 2-D weighted histogram of attenuation-delay of all time-frequency bins. The weighting is done by maximum likelihood estimator, $|\hat{x}_1(\omega,\tau)\hat{x}_2(\omega,\tau)|^p\omega^q$, that is proposed in [133]. We experienced that the simple histogram proposed in original DUET[118] does not perform well. We believe that this is because the relative delay and attenuation are small and SNR is relatively low.

Figure 6.5 shows the weighted histogram for three NIOS®-II cores on an FPGA (left) and three cores of a A33-Olimexlino (right). All the cores are running a simple microbenchmarks (i.e., simple loops) which each of them generates a spike on a different frequency. Each peak in the histogram represent a source. The three tallest peaks are due to the on-chip

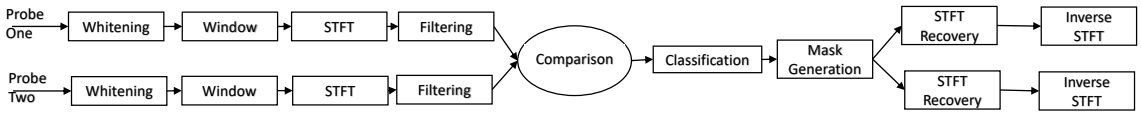


Figure 6.6: The flow graph of separation technique.

cores while the other two peaks are due to the interference from other sources on the chip or component on the board. The peaks corresponding to cores can also have different hight because the number of T-F bins that are affected by a core is different. For example, a core that is running a loop with lower frequency causes more harmonics and hence affects more T-F bins. Hence, one peak can be an order of magnitude larger then other peaks because it has more spikes in its spectral footmarks. As the histogram suggests, the challenge in the separation of sources with extreme proximity is that the delay between them can be very small. However, we were still able to distinguish them by right weighting coefficients and increasing the number of frequency bins in which we calculate the STFT.

After identification of the peaks, we generate the T-F masking matrix M in equation 6.4 by simply assigning each time-frequency bin to the source which its peak is the closest. Then we add the T-F bins associated to clock and T-F masks of any other interference that is not the result of core activity to the T-F mask of each core. We form the time-frequency representation of estimated output by (6.5). In the last stage, we reconstruct the time domain signal of the sources by calculating the inverse short-time Fourier transform.

Figure 6.4 shows the result of this method. Two benchmarks from Mibench suite [99], Basicmath and Bitcount, are running on two cores of A33-Olimexlino. On the left hand the input from one of the input probes is presented. Each superimposed channel has frequency signature of both benchmarks. On the right hand side, we can see the separated signals by the algorithm. The Bitcount's frequency footmarks on the top, is distinct from FFT's recovered signal on the bottom. These two snippets accurately match the signature of each application when we run them in absence of any other application. Bitcouns and Basicmath are largely ω-disjoint orthogonal. The frequencies that carry most information (i.e., spikes)

at each time-frequency bin are different across two benchmarks and do not overlap.

6.5 Experimental Results

We applied the time-frequency masking method mentioned in Section 6.4.2 to two scenarios. First, we separated EM emanation signals generated by two on-chip cores, on an FPGA chip and an off-the-shelf IoT device. Then, we used the same approach to separate EM radiation by two IoT boards.

6.5.1 Multi-core

For our experimental setup, we used an A33-Olinuxino IoT board [136]. This board has a quad core Cortex-A7 [137] running at $1.08GHz$. We used Linux affinity command to assign each benchmark to a core and prevent tasks from switching to another core. In all experiments with two active cores, we used core number one and four from Figure 6.3. The experiment with three benchmarks used core number three as well. We also implemented two NIOS®-II cores on an Altera Cyclone V FPGA using a Terrasic DE0-CV development board. Each core operates at $138MHz$. This clock rate is chosen solely because it was the highest clock rate that was not perturbed with high interference from other components on the board. All the memory is on-chip to make sure there is no off-chip activity and all the EM leakage comes from the chip itself. We used a dual-channel software-defined radio (USRP B210 [138]) and two identical hand-made probes. Placement of the probes is done by trial-and-error to see a sufficient relative attenuation for the algorithm to operate. However, the location of the probes are fixed through all executions. We recorded the signal over a $10MHz$ bandwidth. The setup for this measurement is shown in Figure 6.7. We executed five groups of benchmarks containing four benchmarks from Mibench[99] suite. Each experiment contains two benchmarks except the last one which contains 3 benchmarks. Unfortunately, we could not fit all three benchmarks on the FPGA's on-chip memory and this benchmark trio was only ran on the Olinuxino device.

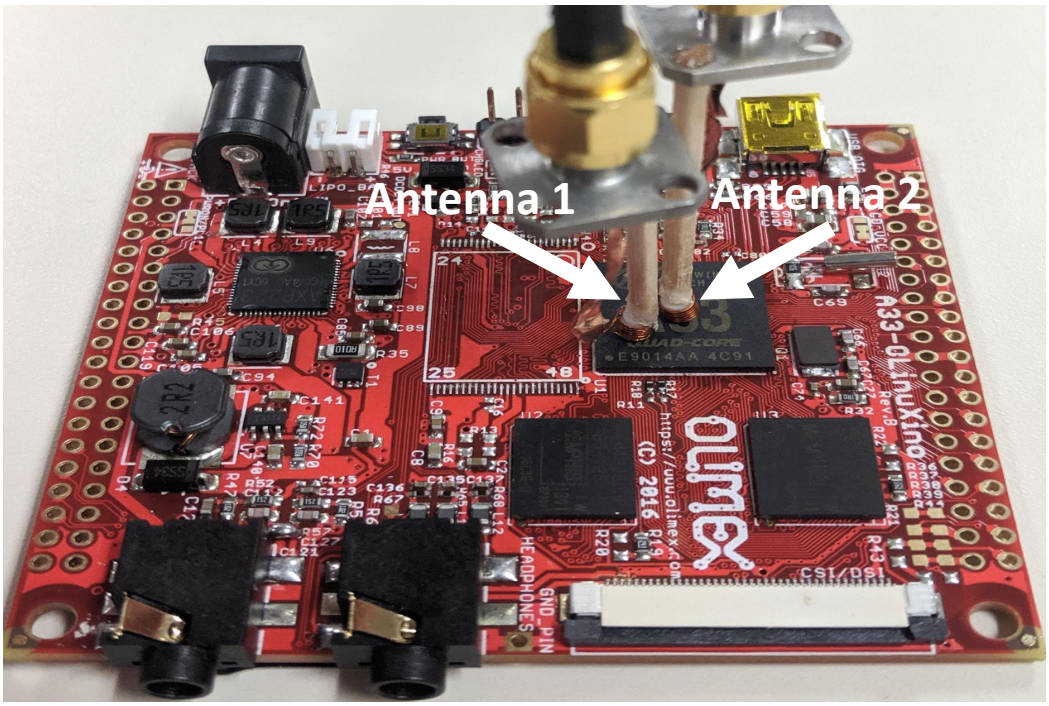


Figure 6.7: The experimental setup with A33-Olinuxino and two handmade antennas.

First, we run each benchmark separately, in absence of any other benchmark. These signals are the ground-truth and we use them as references. Then we run each microbenchmark pair and record the signals synchronously from two channels. We follow the flow shown in Figure 6.6 and apply the separation algorithm as we discussed in 6.4.2. We use window size of $1ms$. To evaluate the separation technique, we compare the separated signals against their corresponding reference signals. We measure the accuracy of classification of time-frequency spikes to the program that produced them. We define accuracy as the percentage of T-F bins in the reference signal that match the corresponding T-F bin in the reconstructed signal. We also define *False Positive* rate as the percentage of T-F bins in the reconstructed signal that are absent in the reference signal. False positive rate represents the remaining interference in the signal.

In figure 6.8 a comparison between a snippet of the reconstructed FFT and the reference signature of FFT is presented. The FFT signal is separated from the execution of FFT and Bitcounts benchmark pair on the A33-Olinuxino device. All the spikes in blue

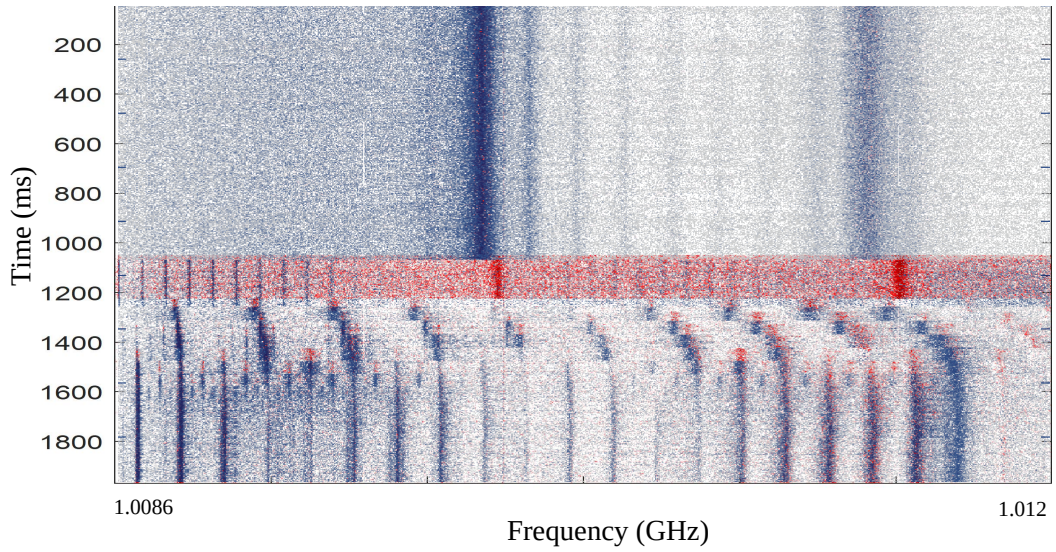


Figure 6.8: Comparison between the reference signature of FFT and the reconstructed FFT signal from FFT+Bitcounts. The spikes in blue are accurately reconstructed. The spikes in red are false positives for Bitcounts.

color are correctly matched with the reference signature. These spikes are reflected in the accuracy. However, the spikes shown in red are incorrectly assigned to the other output channel/application (Bitcounts). These spikes will increase the false positive for Bitcounts.

Table 6.1 presents the results for accuracy and false positive rate. For the A33-Olinuxino device, on average, the proposed technique could separate superimposed signals by classifying spikes of the input signal with 89% accuracy and 9%false positive (i.e., remaining interference). The lowest accuracy and false positive belongs to the superposition of Basicmath and FFT. Most of the inaccuracy and false positive comes from a section of FFT's signal that was completely misclassified. The histogram of this benchmark pair has more than two peaks and multiple peaks were representing FFT. We could not reduce them to two peaks even by changing the weighting coefficients. On the FPGA device, the accuracy is 77%, on average, and false positive rate is 20%. Overall, The separation algorithm could recover signals from A33-Olinuxino with higher accuracy. This is mainly because A33-Olinuxino usually generates sharper spikes in spectrum and, hence, spectral footmarks are more unlikely to overlap.

Table 6.1: Accuracy(%) and false positive(%) results of source separation of on-chip cores.

	Olinuxino		FPGA	
	Accuracy(%)	False Positive(%)	Accuracy(%)	False Positive(%)
Bitcounts	87	15	86	9
FFT	96	2	95	1
Bitcounts	89	13	81	34
Basicmath	97	1	95	13
Basicmath	92	21	54	32
FFT	74	10	60	18
SHA	85	7	74	35
Basicmath	94	1	83	19
SHA	91	15	-	-
Bitcounts	97	3	-	-
Basicmath	89	7	-	-

Lastly, we successfully separate signals from a superposition of three applications. Most of the false positives were due the remaining mutual interference from Basicmath and SHA. These two were executed on core one and three. We suspect that the radiation sources of these two cores are physically closer to each other and that makes the separation more challenging.

6.5.2 Two devices

One of the main advantages of exploiting EM side-channel analysis is that it can be performed at a distance [120]. Suppose a realistic scenario in which there are more than one target electronic device (e.g., Computer, IoT devices, etc.) in proximity of each other (e.g., server rack). Signal is collected from each device by a dedicated directive antenna. Using directive antenna with a focused, narrow radiowave beam width, permits more precise targeting of radiation. However, depending on the proximity of the elements in the setup, each receiving channel receives signal from All components on the device, and it may also receive interference from other devices. However, the same technique is able to separate signals that are produced by each device.

Figure 6.9 shows an example setup in which two IoT devices (A13-OLinuXino-MICRO

[51]) are placed in D_s distance of each other. This device uses a single core ARM cortex A8 processor[74]. Two horn antennas are placed in D_a distance of each board. Signals are acquired by a two-channels software-defined radio (SDR) [138]. The experiment has been done for $D_a = D_s = 30cm$ and $15cm$. Two benchmarks from Mibench suite (*Bitcounts* and *FFT*) are selected. We record reference signals when only one device is executing a benchmark and the other device is off.

Table 6.5.2 presents the results of the accuracy and false positive in this scenario. We could separate the source signals with 93% accuracy and 5% false positive rate. We could not find any significant difference in performance of the algorithm by changing the distance. However, the accuracy was improved compared to the multi-core scenario. This is simply because the relative attenuation and delay are larger and that makes the classification of T-F bins easier. Another observation is that we do not see any significant difference between relative delay and attenuation of spikes produced by different components on the device. Hence, all the frequency spikes from a board are classified as one source. Last observation is that in this scenario we are also able to separate signals when both similar devices run two instances of Bitcounts benchmark. This is possible due to the slight difference between clock frequencies of two similar processors (about $100kHz$). Since Bitcounts has sharp spikes in its spectrum, two identical signatures with slight shift appear in the superposition. Hence, the separation algorithm can still perform well. The inaccuracy comes from a section of the code that generates overlapping footmarks.

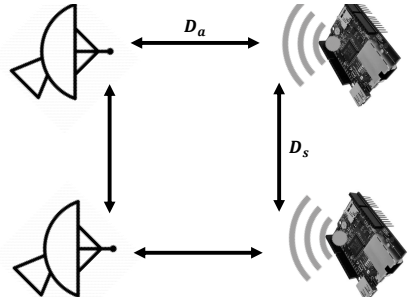


Figure 6.9: Experimental setup for mutual interference between two boards.

Table 6.2: Accuracy(%) and false positive(%) results of source separation of two devices.

	$D_a = D_s = 15cm$		$D_a = D_s = 30cm$	
	Accuracy	False Positive	Accuracy	False Pasitive
Bitcounts	91	4	91	5
FFT	93	1	92	1
Bitcounts	94	7	96	3
Basicmath	97	2	92	1
Bitcounts	91	3	91	2
Bitcounts	89	2	93	2

6.5.3 Use case: Malware detection

To demonstrate how previous studies can benefit from our suggested technique, we provide a simple use-case. As we explained before, few recent studies used the EM spectral signature of software applications for the purpose of malware detection. We devise a simple malware detector that can operate on a single-core system. The source separation block enables the same malware detector, with no required change or retraining, to monitor applications when multiple benchmarks are running. This malware detector is a simplified version of [109, 139] and is implemented as follow. The normal behavior of an application is represented by executing it individually, in absence of any other benchmark. We record the signal and transform it into time-frequency domain. For each window, we collect the frequencies in which five highest spikes occurred. To monitor the new runs of the same application, we repeat this process and compare the frequencies in which the spikes happened. If the difference between them was less than a threshold, the execution is normal. If more than two spikes varied from the reference frequency list for more than 30 consecutive windows, it is classified as anomalous.

We train this malware detector for SHA benchmark on an A33-Olinuxino device. We run the applications multiple times and monitor it by comparing it to the training data. Malware detector has zero false alarms. Then we inject 8 instructions inside the body of the longest loop. The malware detector raises the alarm as soon as it enters the anomalous loop.

However, the same malware detector fails to monitor SHA when a group of SHA, Basicmath, and Bitcounts run on three cores of the same chip. The malware detector falsely raises the red flag for 100% of the execution time. This is simply because the other two applications cause spikes in the spectrum and create misclassification. To solve this problem, we then apply the separation technique in advance. The source separation block can provide an output that contains spikes corresponding to SHA with 91% accuracy and 15% false positive. We then apply the malware detector on this new signal. This results in only 7% false alarm in the duration of execution and 100% of detection accuracy when anomalous version is executed.

6.5.4 ω-disjoint orthogonality

There are two sources of inaccuracy in the results that we presented in this section. First, it is, partially, due to the process of generating masks. Since the relative delay and attenuation are very small, T-F bins may be assigned to wrong source due to the classification error. As we saw in Section 6.5.2, the accuracy improves as the distance between sources significantly increases. However, the accuracy does not increase any further by increasing the distance. The second source of inaccuracy is due to the underlying assumption of binary masking that only one source has a spike in each T-F bin. Since the time-frequency footmarks of applications may overlap, some level of inaccuracy is inevitable. We have already seen that T-F masking can be effective in separating software application's spikes. However, in this section we quantify that how much two applications' EM signatures are ω-disjoint orthogonality if the binary masks could be generated by an ideal classification algorithm.

We only use the reference signals to generate ideal masks. First, we calculate time-frequency representation of each reference signal by using a 1 millisecond windows. Then, we lay out a T-F binary mask that presents only the spikes by simple thresholding. This provides a mask for each benchmark that is ideal for that benchmark and can preserve all of

Table 6.3: Measurement of ω-disjoint orthogonality between benchmark pairs, given an ideal masks for each benchmark.

	FPGA		Olinuxino	
	Mask 1	Mask 2	Mask 1	Mask 2
Bitcounts+FFT	90	98	93	99
Bitcounts+Basicmath	90	99	97	99
Basicmath+FFT	78	90	89	93
SHA+Basicmath	99	98	99	98

its spikes. For each superimposed signal due to execution of two benchmark pair, we can perfectly separate one of the benchmarks by its ideal mask. We have to use the complement binary mask to recover the other source signal. We calculate the orthogonality metric for each of these two ideal masks by equation (6.7).

The result is presented in Table 6.5.4. The spectral frequency suggests that most of the application pairs have high degree of ω-disjoint orthogonality because their orthogonality numbers are close to one. Hence, a right T-F masking can achieve almost perfect recovery of at least one of the benchmarks. It also shows the inherent inaccuracy that is due to binary masking that an ideal mask for one benchmark causes an imperfect mask for the other benchmark. As Table 6.5.4 suggests, the lowest number belongs to the pair of Basicmath and FFT. This is because these two signals have spectral signatures that are more spread out, and it is more likely that they overlap. Another observation is that the orthogonality is lower on FPGA than Olinuxino. This is because FPGA has a much lower clock-rate and that causes spikes on lower frequencies with more harmonics which increases the chance of an overlap between two time-frequency footmarks.

6.6 Summary

In this chapter we presented a technique for source separation of EM side-channel signals from a pair of signals that are superposition of multiple AM-modulated signals. Many previously proposed methods which analyze side-channel signals, such as malware detector and profiling, can be employed directly on the recovered signals by this technique. This

method can separate EM signals radiated from multiple on-chip processing cores. It can also cancel the mutual interference of EM emanation signals from multiple devices.

We suggested using a time-frequency masking technique that takes advantage of disparity between radiation pasterns of different sources. We illustrated binary time-frequency masks can accurately recover signals since applications' generate disjoint time-frequency spike patterns. To produce these masks we used two input antennas. We classified time-frequency bins based on their relative attenuation and delay. We applied this technique to separate spectral footmarks generated by execution of two and three applications on a multi-core ARM-based processor. We successfully separated source signals with 89% accuracy. Similarly we were able to separate signals from two on-chip cores on an FPGA as accurate as 77%. Lastly, we applied this method to separate mutual interference of two devices with 15 and $30cm$ distance. We separated these signals with 93% accuracy and 5% false positive rate. Finally, we showed how this separation algorithm can help an existing malware detector to work in a multi-core scenario without any change or retraining.

CHAPTER 7
RESEARCH CONTRIBUTIONS AND FUTURE WORK

7.1 Research Contributions

This research developed methods for profiling applications via EM emanations from computing devices. These EM emanations were previously studied for security purposes, for example to study how EM emanations can potentially be used to extract secret keys in cryptography. We have demonstrated that EM emanations are information-rich signals. Hence, it is viable to use them to profile software programs in large granularity to find *hot spots* or to monitor architectural event as small as LLC miss. Moreover, we showed that this information can be used for anomaly detection for embedded processors such those used in IoT devices. Lastly, we provided a method that can extend the use of these methods to more sophisticated processors.

The research contributions of this work are:

1. Spectral Profiling, a new method for profiling program execution without instrumenting or otherwise affecting the profiled system [22]. Spectral Profiling monitors EM emanations unintentionally produced by the profiled system, looking for spectral ”spikes” produced by periodic program activity (e.g. loops). This allows Spectral Profiling to determine which parts of the program have executed at what time. By analyzing the frequency and shape of the spectral ”spike”, Spectral Profiling can obtain additional information such as the per-iteration execution time of a loop. The key advantage of Spectral Profiling is that it can monitor a system as-is, without program instrumentation, system activity, etc. associated with the profiling itself, i.e. it completely eliminates the ”Observer’s Effect” and allows profiling of programs whose execution is performance-dependent and/or programs that run on even the simplest

embedded systems that have no resources or support for profiling. We evaluated the effectiveness of Spectral Profiling by applying it to several benchmarks from MiBench suite on a real system, and also on a cycle-accurate simulator. Our results confirm that Spectral Profiling yields useful information about the runtime behavior of a program, allowing Spectral Profiling to be used for profiling in systems where profiling infrastructure is not available, or where profiling overheads may perturb the results too much ("Observer's Effect").

2. EM-Based Detection of Deviations in Program Execution (EDDIE) [109, 140, 141], a new method for detecting anomalies in program execution, such as malware and other code injection, without introducing any overheads, adding any hardware support, changing any software, or using any resources on the monitored system itself. During training, EDDIE characterizes normal execution behavior in terms of peaks in the EM spectrum that are observed at various points in the program execution, but it does not need any characterization of the virus or other code that might later be injected. During monitoring, EDDIE identifies peaks in the observed EM spectrum, and compares these peaks to those learned during training. We evaluated EDDIE on a real IoT system and in a cycle-accurate simulator, and find that even relatively brief injected bursts of activity (a few milliseconds) were detected by EDDIE with high accuracy, and that it also accurately detects when even a few instructions are injected into an existing loop withing the application.

3. EMPROF, a novel method that leverages EM emanations of the profiled system's processor to profile its memory performance without any support on, or interference with, the profiled system [142]. Specifically, EMPROF analyzes the system's EM emanations to identify processor stalls associated with last-level cache (*LLC*) misses. It is able to accurately pinpoint LLC misses in the execution timeline, and also measure the cost (stall time) of each of these misses. EMPROF has zero "ob-

server effect", so it can be used to profile applications that adjust their activity to their performance. It has no overhead on target machine, so it can be used for profiling embedded, hand-held, and IoT devices which usually have limited support for collecting, and limited resources for storing, the profiling data. Finally, since EMPROF can profile the system as-is, its profiling of boot code and other hard-to-profile software components is as accurate as its profiling of application code. To illustrate the effectiveness of EMPROF, we first validate its results using micro-benchmarks with known memory behavior, and also on SPEC benchmarks running a cycle-accurate simulator that can provide detailed ground-truth data about LLC misses and processor stalls. We then demonstrate the effectiveness of EMPROF on real systems, including profiling of boot activity, show how its results can be attributed to the specific parts of the application code when that code is available, and provide additional insight on the statistics reported by EMPROF and how they are affected by the EM signal bandwidth provided to EMPROF.

4. Extending Spectral Profiling and EDDIE to multi-core processors by source separation of EM side-channels from signals that are *superposition* of multiple AM-modulated signals. We achieved source separation by classifying spectral "spikes" of superimposed signals between sources. We evaluated the effectiveness of this method to separate EM emanation signals when multiple benchmarks run on a multi-core processor. We evaluate this technique on a multi-core FPGA board and an off-the-shelf multi-core Internet-of-Things (IoT) device. For the IoT device, time-frequency spikes of the separated signals matched the original signal (i.e., solo execution) with 89% accuracy and only 9% false positive. Further, we achieve 77% accuracy and 20% false positive rate on the FPGA device. We also showed this technique can successfully separate mutual EM interference when multiple IoT devices are located close to each other. In this case we achieved 94% accuracy and 5% false positive rate. Finally, we demonstrated that an existing malware detector can benefit from source

separation to work on a multi-core processor without any change or retraining.

7.2 Future Research Directions

The greatest opportunity for future direction of this work is to improve and expand EMPROF. Currently, EMPROF enables software developers to externally profile cache misses. However, performance optimization and debugging usually requires measurement of many more profiling metrics from the target device. The same proposed approach can be adopted to measure other events. Two categories of events are suitable to be targeted in the future work. First, external profiling of information that is usually provided by hardware counters, such as microarchitectural events and hardware interrupts, can benefit embedded devices. Same as cache profiling, recording these counters with high resolution introduces observers effect. Second, profiling software events initiated by Operating System(OS), i.e. system calls, is valuable. Profiling such events is done with relatively low cost and high accuracy by the OS. However, profiling such events is necessary to accurately track applications. A software application may get interrupted by OS for IO access, privilege control, etc.. This also enables the previous methods to truly support multiprogramming where more than one software application is running. A profiler that can target both software and hardware events may also integrate proposed Spectral Profiling as well to achieve high resolution code attribution.

Another opportunity of future research lies in possible improvements to EDDIE. Currently EDDIE uses a simple statistical approach for malware detection. More accurate modeling, e.g. recurrent neural networks, may improve its performance significantly. Moreover, detection accuracy may improve by even more sophisticated model that integrates software/hardware profiling measurements from previously discussed framework.

In addition to the future research possibilities for improvement to proposed profiling and malware detection, there are many opportunities in signal processing. Currently, Spectral Profiling does not support Spread-Spectrum Clock Generation (SSCG) which is used in processors with high clock rate. SSCG reduces the spectral density of EM interference by intentionally modulating the ideal position of the clock edge such that the resulting signals spectrum is "spread", around the ideal frequency of the clock. An additional "de-spreading" stage is needed before any other EM analysis. Noise and interference canceling and increase in SNIR of inputs by filtering techniques or better probing can improve the performance of all proposed methods.

Further improvement in blind source separation of multi-core components is another major research direction. While we demonstrated source separation is feasible, accurate source separation for larger number of cores needs further signal processing. An improvement to proposed mask generation technique and additional investigation on statistical methods, e.g. Independent Component Analysis, can benefit many multi-core platforms.

REFERENCES

[1] H. J. Highland, "Electromagnetic radiation revisited," *Computers and Security*, pp. 85–93, Dec. 1986.

[2] D Agrawal, B Archambeult, J. R. Rao, and P Rohatgi, "The EM side-channel(s)," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002*, 2002, pp. 29–45.

[3] A. Zajić and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 56, no. 4, pp. 885–893, 2014.

[4] K Gandolfi, C Mourtel, and F Olivier, "Electromagnetic analysis: concrete results," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2001*, 2001, pp. 251–261.

[5] M. G. Khun, "Compromising emanations: eavesdropping risks of computer displays," *The complete unofficial TEMPEST web page: http://www.eskimo.com/˜joelm/tempest.html*, 2003.

[6] H. Sekiguchi and S. Seto, "Measurement of radiated computer rgb signals," *Progress in Electromagnetic Research C*, pp. 1–12, 2009.

[7] Y. Suzuki and Y. Akiyama, "Jamming technique to prevent information leakage caused by unintentional emissions of pc video signals," in *Electromagnetic Compatibility (EMC), 2010 IEEE International Symposium on*, 2010, pp. 132–137.

[8] M. G. Kuhn, "Compromising emanations of lcd tv sets," *IEEE Transactions on Electromagnetic Compatibility*, pp. 564–570, 2013.

[9] T Plos, M Hutter, and C Herbst, "Enhancing side-channel analysis with low-cost shielding techniques," in *Proceedings of Austrochip*, 2008.

[10] F. Poucheret, L. Barthe, P. Benoit, L. Torres, P. Maurine, and M. Robert, "Spatial EM jamming: A countermeasure against EM Analysis?" In *Proceedings of the 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC)*, 2010, pp. 105–110.

[11] H. Tanaka, "Information leakage via electromagnetic emanations and evaluation of Tempest countermeasures," in *Lecture notes in computer science, Springer*, 2007, pp. 167–179.

[12] Y. ichi Hayashi, N. Homma, T. Mizuki, H. Shimada, T. Aoki, H. Sone, L. Sauvage, and J.-L. Danger, "Efficient evaluation of em radiation associated with information leakage from cryptographic devices," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 555–563, 2013.

[13] H. Sekiguchi and S. Seto, "Study on maximum receivable distance for radiated emission of information technology equipment causing information leakage," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 547–554, 2013.

[14] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *Presented as part of the 2013 USENIX Workshop on Health Information Technologies*, 2013.

[15] S. S. Clark, B. Ransford, and K. Fu, "Potentia est scientia: Security and privacy implications of energy-proportional computing," in *7th USENIX Workshop on Hot Topics in Security, HotSec'12, Bellevue, WA, USA, August 7, 2012*, 2012.

[16] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys 2008), Breckenridge, CO, USA, June 17-20, 2008*, 2008, pp. 239–252.

[17] R. Callan, F. Behrang, A. G. Zajic, M. Prvulovic, and A. Orso, "Zero-overhead profiling via EM emanations," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 401–412.

[18] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, 2011.

[19] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 4th International Conference on Computing Frontiers*, ser. CF '07, 2007.

[20] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, 2012.

[21] Y. Sato, Y. Inoguchi, and T. Nakamura, "On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11, 2011.

[22] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, IEEE, 2016, pp. 1–11.

[23] J Young, "How old is Tempest?" *Online response collection, http://cryptome.org/tempest-old.htm*, 2000.

[24] W. van Eck, "Electromagnetic radiation from video display units: an eavesdropping risk?" *Computers and Security*, pp. 269–286, Dec. 1985.

[25] Y. ichi Hayashi, N. Homma, T. Mizuki, T. Aoki, H. Sone, L. Sauvage, and J.-L. Danger, "Analysis of electromagnetic information leakage from cryptographic devices with different physical structures," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 571–580, 2013.

[26] D. Genkin, I. Pipman, and E. Tromer, "Get your hands of my laptop: Physical side-channel key-extraction attacks on pcs," in *Proceedings of Cryptographic Hardware and Embedded Systems*, ser. CHES '14', 2014.

[27] R. Callan, A. Zajić, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014.

[28] B. Durak, *Controlled CPU TEMPEST Emanations*, 1999.

[29] S. Clark, H. Mustafa, B. Ransford, J. Sorber, K. Fu, and W. Xu, "Current events: Identifying webpages by tapping the electrical outlet," in *Computer Security ESORICS 2013*, ser. Lecture Notes in Computer Science, J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134, Springer Berlin Heidelberg, 2013, pp. 700–717, ISBN: 978-3-642-40202-9.

[30] C. Aguayo Gonzlez and J. Reed, "Power fingerprinting in sdr integrity assessment for security and regulatory compliance," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, pp. 307–327, 2011.

[31] R. Callan, N. Basta, A. Zajic, and M. Prvulovic, "A new approach for measuring electromagnetic side-channel energy available to the attacker in modern processor-memory systems," in *Proceedings of the 9th European Conference on Antennas and Propagation (EuCAP)*, 2015.

[32] R. Callan, A. Zajic, and M. Prvulovic, "FASE: Finding Amplitude-modulated Side-channel Emanations," in *the 42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[33] R. Callan, N. Popovic, A. Daruna, E. Pollmann, A. Zajic, and M. Prvulovic, "Comparison of electromagnetic side-channel energy available to the attacker from different computer systems," in *Proceedings of the 2015 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2015.

[34] M. R. Guthaus, J. S. Pingenberg, D. Emst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01, 2001.

[35] E. Sejdić, I. Djurović, and J. Jiang, "Time–frequency feature representation using energy concentration: An overview of recent advances," *Digit. Signal Process.*, vol. 19, no. 1, pp. 153–183, Jan. 2009.

[36] S. Debray and W. Evans, "Profile-guided code compression," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02, 2002.

[37] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "HOLMES: EFFECTIVE STATISTICAL DEBUGGING VIA EFFICIENT PATH PROFILING," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009.

[38] M. D. Ernst, J. CockrelI, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99, 1999.

[39] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" In *ACM Trans- actions on Computer Systems*, ser. TOCOS, 1997.

[40] Intel-Corporation, *Intel vtune amplifier*, `https://software.intel.com/en-us/intel-vtune-amplifier-xe/details`, accessed April 2, 2016.

[41] X. Yang, S. M. Blackburn, and K. S. McKinley, "Computer performance microscopy with shim," in *ACM/IEEE International Symposium on Computer Architecture*, ser. ISCA '15, 2015.

[42] S. L. Graham, P. B. Kessler, and M. K. McKusiclc, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82, 1982.

[43] Linux, *Linux kernel profiling with perf*, `https://perf.wiki.kernel.org/index.php/Main_Page`, accessed April 3, 2016.

[44] Q. Zhao, I. Cutcutache, and W. Wong, "Pipa: Pipelined profiling and analysis on multi-core systems," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08, 2008.

[45] S. Wallace and K. Hazelwood, "Superpin: Parallelizing dynamic instrumentation for real-time performance," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07, 2007.

[46] Intel-Corporation, *Precise-event-based-sampling on intel processors*, `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf`, accessed April 2, 2016.

[47] ——, *Intel performance counter monitor*, `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`, accessed April 2, 2016.

[48] ARM, *Arm performance monitor unit*, `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/Bcgddibf.html`, accessed April 2, 2016.

[49] G. Paoloni, "White paper: How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures," Intel Corporation, Tech. Rep., 2010.

[50] Keysight-Technologies, *Data-sheet: Probing solutions for logic analyzers*, `http://literature.cdn.keysight.com/litweb/pdf/5968-4632E.pdf`, accessed April 5, 2016.

[51] Olimex, *A13-olinuxino-micro user manual*, `https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware`, accessed April 3, 2016.

[52] T. Ball and J. R. Larus, "Efficient Path Profiling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO '96, 1996.

[53] M. D. Bond and K. S. McKinley, "Practical path profiling for dynamic optimizers," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '05, 2005.

[54] K. Vaswani, A. V. Nori, and T. M. Chilimbi, "Preferential path profiling: Compactly numbering interesting paths," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07, 2007.

[55] R. Joshi, M. D. Bond, and C. Zilles, "Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems," in *Proceedings of the Interna-*

tional Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, ser. CGO '04, 2004.

[56] S. Baswana, S. Roy, and R. Chouhan, "Pertinent path profiling: Tracking interactions among relevant statements," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13, 2013.

[57] M. Afraz, D. Saha, and A. Kanade, "P3: Partitioned path profiling," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015.

[58] Q. Wu, A. Pyatakov, A. Spiridonov, E. Raman, D. W. Clark, and D. I. August, "Exposing memory access regularities using object-relative memory profiling," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, 2004, p. 315.

[59] P. Nagpurkar, H. Mousa, C. Krintz, and T. Sherwood, "Efficient remote profiling for resource-constrained devices," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 3, no. 1, pp. 35–66, 2006.

[60] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A tool suite for simulation based analysis of memory access behavior," in *International Conference on Computational Science*, Springer, 2004, pp. 440–447.

[61] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the mips r10000 performance counters," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, 1996, p. 16.

[62] P. Agrawal, A. J. Goldberg, and J. A. Trotter, *Interrupt-based hardware support for profiling memory system performance*, US Patent 5,768,500, 1998.

[63] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Supercomputing, ACM/IEEE 2000 Conference*, IEEE, 2000, pp. 42–42.

[64] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff, "Performance measurement intrusion and perturbation analysis," *IEEE Transactions on parallel and distributed systems*, vol. 3, no. 4, pp. 433–450, 1992.

[65] A. D. Malony and S. S. Shende, "Overhead compensation in performance profiling," in *European Conference on Parallel Processing*, Springer, 2004, pp. 119–132.

[66] V. M. Weaver, "Self-monitoring overhead of the linux perf_ event performance counter interface," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, IEEE, 2015, pp. 102–111.

[67] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, ACM, vol. 40, 2005, pp. 190–200.

[68] M. Martonosi, A. Gupta, and T. Anderson, "Memspy: Analyzing memory system bottlenecks in programs," in *ACM SIGMETRICS Performance Evaluation Review*, ACM, vol. 20, 1992, pp. 1–12.

[69] T. Mudge, "Power: A first-class architectural design constraint," *Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[70] M. K. Gowan, L. L. Biro, and D. B. Jackson, "Power considerations in the design of the alpha 21264 microprocessor," in *Proceedings of the 35th annual Design Automation Conference*, ACM, 1998, pp. 726–731.

[71] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, "Predicting performance impact of dvfs for realistic memory systems," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2012, pp. 155–165.

[72] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, IEEE, 2004, pp. 338–349.

[73] S. Eyerman and L. Eeckhout, "A counter architecture for online dvfs profitability estimation," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1576–1583, 2010.

[74] ARM, *Arm cortex a8 processor manual*, `https://www.arm.com/products/processors/cortex-a/cortex-a8.php`, accessed April 3, 2016.

[75] J. Renau, L. Ceze, J. Tuck, M. Prvulovic, and M. C. Huang, *SESC*, 2006.

[76] AARONIA, *Datasheet: Rf near field probe set dc to 9ghz*, `http://www.aaronia.com/Datasheets/Antennas/RF-Near-Field-Probe-Set.pdf`, accessed April 6, 2016.

[77] G. Reinman and N. Jouppi, "Cacti 2.0: An integrated cache timing and power model," *Technical Report*, 2000.

[78] D. Brooks, V. Tiwari, and M. Martonosi, in *ACM/IEEE International Symposium on Computer Architecture*, ser. ISCA-27, 2000, pp. 83–94.

[79] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*, ACM, 2008, pp. 15–26.

[80] M. Alazab, S. Venkatraman, P. Watters, and M. Alazab, "Zero-day malware detection based on supervised learning algorithms of api call signatures," in *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121*, Australian Computer Society, Inc., 2011, pp. 171–182.

[81] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware-assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, 2015, p. 74.

[82] M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh, "Hardware-assisted detection of malicious software in embedded systems," *IEEE Embedded Systems Letters*, vol. 4, no. 4, pp. 94–97, 2012.

[83] M. Zhang and R Sekar, "Control flow integrity for cots binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[84] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[85] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI, Boston, MA, USA: ACM, 2004, pp. 85–96, ISBN: 1-58113-804-0.

[86] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[87] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[88] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, IEEE, 2008, pp. 173–184.

[89] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*, ACM, 2008, pp. 51–62.

[90] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, IEEE, 2008, pp. 233–247.

[91] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, ser. DSN '12, 2012.

[92] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 447–462.

[93] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, IEEE Computer Society, 2005, pp. 178–183.

[94] R. G. Ragel and S. Parameswaran, "Impres: Integrated monitoring for processor reliability and security," in *Proceedings of the 43rd annual Design Automation Conference*, ACM, 2006, pp. 502–505.

[95] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," in *Proceedings of the 44th annual Design Automation Conference*, ACM, 2007, pp. 483–488.

[96] C. Schmidt, *Low Level Virtual Machine (LLVM)*, `https://github.com/llvm-mirror/llvm`, 2014.

[97] F. J. M. Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78,

[98] Keysight-Technologies, *Dsos804a high-definition oscilloscope: 8 ghz, 4 analog channels*, `http://www.keysight.com/en/pdx-x202073-pn-DSOS804A/high-definition-oscilloscope-8-ghz-4-analog-channels?cc=US&lc=eng`.

[99] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, IEEE, 2001, pp. 3–14.

[100] G. Reinman and N. P. Jouppi, "Cacti 2.0: An integrated cache timing and power model," *Western Research Lab Research Report*, vol. 7, 2000.

[101] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00, Vancouver, British Columbia, Canada: ACM, 2000, pp. 83–94, ISBN: 1-58113-232-8.

[102] Hynix, *2gb ddr3 sdram lead-free and halogen-free (rohs compliant) h5tq2g63bfr*, http://www.farnell.com/datasheets/1382727.pdf?_ga=2.38941163.99566001.1522095243-708330394.1522095243, "Rev 0.5", accessed March 20, 2018.

[103] Samsung, *Samsung galaxy centura sch-s738c user manual with specs*, https://www.cnet.com/products/samsung-galaxy-centura-sch-s738c-3g-4-gb-cdma-smartphone/specs/, accessed March 26, 2018.

[104] Alcatel, *Alcatel ideal / streak specifications*, https://www.devicespecifications.com/en/model/e2ef3d31, accessed March 26, 2018.

[105] Keysight, *N9020a mxa spectrum analyzer*, https://www.keysight.com/en/pdxx202266-pn-N9020A/mxa-signal-analyzer-10-hz-to-265-ghz?cc=US&lc=eng, accessed March 26, 2018.

[106] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[107] ThinkRF, *Rtsa v3 real-time spectrum analyzer*, https://s3.amazonaws.com/ThinkRF/Documents/ThinkRF+RTSAv3+TDS+74-0034.pdf, accessed March 26, 2018.

[108] Signatec, *Px14400a 400 ms/s, 14 bit, ac coupled, 2 channel, xilinx virtex-5 fpga, pcie x8, high speed digitizer board*, http://www.signatec.com/products/daq/high-speed-fpga-pcie-digitizer-board-px14400.html, accessed March 26, 2018.

[109] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "Eddie: Em-based detection of deviations in program execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture. ACM*, 2017, pp. 333–346.

[110] A. Zajic and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 56, no. 4, pp. 885–893, 2014.

[111] S. Haykin and Z. Chen, "The cocktail party problem," *Neural computation*, vol. 17, no. 9, pp. 1875–1902, 2005.

[112] N. Sehatbakhsh, A. Nazari, A. G. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016*.

[113] V. Carlier, H. Chabanne, E. Dottax, and H. Pelletier, "Electromagnetic side channels of an fpga implementation of aes," in *CRYPTOLOGY EPRINT ARCHIVE, REPORT 2004/145*, Citeseer, 2004.

[114] E. De Mulder, P. Buysschaert, S. Ors, P. Delmotte, B. Preneel, G. Vandenbosch, and I. Verbauwhede, "Electromagnetic analysis attack on an fpga implementation of an elliptic curve cryptosystem," in *EUROCON 2005-The International Conference on" Computer as a Tool"*, IEEE, vol. 2, 2005, pp. 1879–1882.

[115] T. Espitau, P.-A. Fouque, B. Gérard, and M. Tibouchi, "Side-channel attacks on bliss lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 1857–1874.

[116] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. Callan, A. Zajic, and M. Prvulovic, "One&done: A single-decryption em-based attack on openssl's constant-time blinded RSA," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 585–602, ISBN: 978-1-939133-04-5.

[117] R. Rutledge, S. Park, H. Khan, A. Orso, M. Prvulovic, and A. Zajic, "Zero-overhead path prediction with progressive symbolic execution," in *Proceedings of the 41st International Conference on Software Engineering*, IEEE Press, 2019, pp. 234–245.

[118] A. Jourjine, S. Rickard, and O. Yilmaz, "Blind separation of disjoint orthogonal signals: Demixing n sources from 2 mixtures," in *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 00CH37100)*, IEEE, vol. 5, 2000, pp. 2985–2988.

[119] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *International workshop on cryptographic hardware and embedded systems*, Springer, 2001, pp. 251–261.

[120] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The em sidechannel (s)," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2002, pp. 29–45.

[121] A. J. Bell and T. J. Sejnowski, "An information-maximization approach to blind separation and blind deconvolution," *Neural computation*, vol. 7, no. 6, pp. 1129–1159, 1995.

[122] S.-i. Amari, A. Cichocki, and H. H. Yang, "A new learning algorithm for blind signal separation," in *Advances in neural information processing systems*, 1996, pp. 757–763.

[123] J.-F. Cardoso, "High-order contrasts for independent component analysis," *Neural computation*, vol. 11, no. 1, pp. 157–192, 1999.

[124] A. Hyvarinen, "Fast and robust fixed-point algorithms for independent component analysis," *IEEE transactions on Neural Networks*, vol. 10, no. 3, pp. 626–634, 1999.

[125] L. Molgedey and H. G. Schuster, "Separation of a mixture of independent signals using time delayed correlations," *Physical review letters*, vol. 72, no. 23, p. 3634, 1994.

[126] P. Bofill and M. Zibulevsky, "Blind separation of more sources than mixtures using sparsity of their short-time fourier transform," in *Proc. ica*, vol. 2000, 2000, pp. 87–92.

[127] S. Rickard and O. Yilmaz, "On the approximate w-disjoint orthogonality of speech," in *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*, IEEE, vol. 1, 2002, pp. I–529.

[128] S. Schulz and T. Herfet, "On the window-disjoint-orthogonality of speech sources in reverberant humanoid scenarios," in *Proceedings of the 11th International Conference on Digital Audio Effects (DAFx'08)*, 2008, pp. 241–248.

[129] S. Rickard and F. Dietrich, "Doa estimation of many w-disjoint orthogonal sources from two mixtures using duet," in *Proceedings of the Tenth IEEE Workshop on Statistical Signal and Array Processing (Cat. No. 00TH8496)*, IEEE, 2000, pp. 311–314.

[130] T. Melia and S. Rickard, "Underdetermined blind source separation in echoic environments using desprit," *EURASIP Journal on Applied Signal Processing*, vol. 2007, no. 1, pp. 90–90, 2007.

[131] M. Aoki, M. Okamoto, S. Aoki, H. Matsui, T. Sakurai, and Y. Kaneda, "Sound source segregation based on estimating incident angle of each frequency component of input signals acquired by multiple microphones," *Acoustical science and technology*, vol. 22, no. 2, pp. 149–157, 2001.

[132] N. Roman, D. Wang, and G. J. Brown, "Speech segregation based on sound localization," *The Journal of the Acoustical Society of America*, vol. 114, no. 4, pp. 2236–2252, 2003.

[133] O. Yilmaz and S. Rickard, "Blind separation of speech mixtures via time-frequency masking," *IEEE Transactions on signal processing*, vol. 52, no. 7, pp. 1830–1847, 2004.

[134] M. I. Mandel, D. P. Ellis, and T. Jebara, "An em algorithm for localizing multiple sound sources in reverberant environments," in *Advances in neural information processing systems*, 2007, pp. 953–960.

[135] S. Araki, H. Sawada, R. Mukai, and S. Makino, "Underdetermined blind sparse source separation for arbitrarily arranged multiple sensors," *Signal Processing*, vol. 87, no. 8, pp. 1833–1847, 2007.

[136] OLinuXino, *A33-OLinuXino User Manual*, accessed November 10, 2019.

[137] ARM, *Arm cortex a7 processor manual*, accessed April 3, 2016.

[138] B210, *USRP B210 User Manual*, accessed November 10, 2019.

[139] Y. Han, S. Etigowni, H. Liu, S. Zonouz, and A. Petropulu, "Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 1095–1108.

[140] N. Sehatbakhsh, M. Alam, A. Nazari, A. Zajic, and M. Prvulovic, "Syndrome: Spectral analysis for anomaly detection on medical iot and embedded devices," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, 2018, pp. 1–8.

[141] N. Sehatbakhsh, H. Hong, B. Lazar, B. Johnson-Smith, O. Yilmaz, M. Alam, A. Nazari, A. Zajic, and M. Prvulovic, "Syndrome: Spectral analysis for anomaly detection on medical iot and embedded devicesexperimental demonstration."

[142] M. Dey, A. Nazari, A. Zajic, and M. Prvulovic, "Emprof: Memory profiling via em-emanation in iot and hand-held devices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 881–893.